Conception et contrôle de haut niveau pour les systèmes sur puce multiprocesseurs adaptatifs

Xin An

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Xin An

Thèse dirigée par **Eric Rutten**
et codirigée par **Abdoulaye Gamatié**

préparée au sein **INRIA-Grenoble**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

High Level Design and Control of Adaptive Multiprocessor Systems-on-Chip

Thèse soutenue publiquement le **16 octobre 2013**,
devant le jury composé de :

Mr Jean-Marc Faure
LURPA, ENS Cachan, Professeur, Rapporteur
Mr Frédéric Mallet
U. Nice Sophia-Antipolis, Maître de Conférences (HdR), Rapporteur
Mme Dominique Borrione
TIMA, Grenoble, Professeur, Examinatrice
Mr Jean-Philippe Diguet
CNRS Lab-Sticc, Lorient, Directeur de Recherche, Examinateur
Mr Hervé Marchand
INRIA, Rennes, Chargé de Recherche, Examinateur
Mr Jean-François Méhaut
UJF-CEA/LIG, Grenoble, Professeur, Examinateur
Mr Eric Rutten
INRIA, Grenoble, Chargé de Recherche, Directeur de thèse
Mr Abdoulaye Gamatié
CNRS LIRMM, Montpellier, Chargé de Recherche, Co-Directeur de thèse

Abstract

The design of modern embedded systems is getting more and more complex, as more functionality is integrated into these systems. At the same time, in order to meet the computational requirements while keeping a low level power consumption, MPSoCs have emerged as the main solutions for such embedded systems. Furthermore, embedded systems are becoming more and more adaptive, as the adaptivity can bring a number of benefits, such as software flexibility and energy efficiency. This thesis targets the safe design of such adaptive MPSoCs.

First, each system configuration must be analyzed concerning its functional and non-functional properties. We present an abstract design and analysis framework, which allows for faster and cost-effective implementation decisions. This framework is intended as an intermediate reasoning support for system level software/hardware co-design environments. It can prune the design space at its largest, and identify candidate design solutions in a fast and efficient way. In the framework, we use an abstract clock-based encoding to model system behaviors. Different mapping and scheduling scenarios of applications on MPSoCs are analyzed via clock traces representing system simulations. Among properties of interest are functional behavioral correctness, temporal performance and energy consumption.

Second, the reconfiguration management of adaptive MPSoCs must be addressed. We are specially interested in MPSoCs implemented on reconfigurable hardware architectures (i.e., FPGA fabrics), which provide a good flexibility and computational efficiency for adaptive MPSoCs. We propose a general design framework based on the discrete controller synthesis (DCS) technique to address this issue. The main advantage of this technique is that it allows the automatic controller synthesis w.r.t. a given specification of control objectives. In the framework, the system reconfiguration behavior is modeled in terms of synchronous parallel automata. The reconfiguration management computation problem w.r.t. multiple objectives regarding e.g., resource usages, performance and power consumption is encoded as a DCS problem. The existing BZR programming language and Sigali tool are employed to perform DCS and generate a controller that satisfies the system requirements.

Finally, we investigate two different ways of combining the two proposed design frameworks for adaptive MPSoCs. Firstly, they are combined to construct a complete design flow for adaptive MPSoCs. Secondly, they are combined to present how the designed runtime manager by the second framework can be integrated into the first framework so as to perform combined simulations and analysis of adaptive MPSoCs.

Contents

1 Introduction **1**
- 1.1 Trends in Embedded Systems . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Contributions . . . 3
- 1.4 Thesis Overview . . . 5

I State of the Art **7**

2 Multiprocessor Systems-on-Chip (MPSoCs) **9**
- 2.1 General Presentation . . . 9
  - 2.1.1 Processing Elements . . . 10
  - 2.1.2 Memory Architecture . . . 11
  - 2.1.3 Interconnect . . . 12
- 2.2 MPSoC Design . . . 13
  - 2.2.1 A Design Flow Model . . . 13
  - 2.2.2 Design Space Exploration . . . 15
  - 2.2.3 A Survey of Existing Approaches . . . 16
- 2.3 Adaptivity of MPSoCs . . . 19
  - 2.3.1 Motivation for Adaptivity in MPSoCs . . . 19
  - 2.3.2 FPGAs as Implementation Platforms for MPSoCs . . . 19
  - 2.3.3 Adaptivity Management . . . 20
- 2.4 Summary and Discussion . . . 23

3 Models, Languages and Tools for Reactive Systems **25**
- 3.1 Data-Flow Based Modeling Formalisms . . . 26
  - 3.1.1 Kahn Process Network and Synchronous Data Flow . . . 26
  - 3.1.2 Data-Flow Synchronous Languages . . . 27
  - 3.1.3 The UML Profile MARTE and CCSL Language . . . 29
- 3.2 Automata-Based Modeling Formalisms . . . 31
  - 3.2.1 StateCharts . . . 31
  - 3.2.2 Automata-Based Synchronous Languages . . . 31
  - 3.2.3 Heptagon Language . . . 32
  - 3.2.4 Formal Definition of Automata . . . 33
- 3.3 Discrete Controller Synthesis (DCS) . . . 35
  - 3.3.1 General Presentation . . . 35
  - 3.3.2 Control Objectives . . . 36
- 3.4 BZR Synchronous Language and DCS . . . 37
- 3.5 Some Approaches Applying Discrete Control . . . 38
- 3.6 Summary and Discussion . . . 38

II Contributions **41**

4 CLASSY: A High Level Design and Analysis Framework for MPSoCs **43**
4.1 Motivation and Contribution . . . 44
4.2 High-Level Modeling of Adaptive MPSoCs . . . 45
4.3 An Abstract Design Framework for Adaptive Systems . . . 47
4.3.1 Application Behavior . . . 47
4.3.2 Execution Platform Behavior . . . 48
4.3.3 Application Mapping on Platforms . . . 49
4.4 Scheduling and Design Analysis . . . 49
4.4.1 Clock Modeling of System Executions . . . 49
4.4.2 Admissible Scheduling of Applications . . . 50
4.4.3 Scheduling Algorithm . . . 51
4.4.4 Performance Analysis . . . 54
4.5 Design Space Exploration . . . 55
4.6 Implementation and Experiments . . . 57
4.6.1 CLASSY Tool . . . 57
4.6.2 Experimental results . . . 58
4.7 Summary and Discussion . . . 66

5 Discrete Control for Reconfiguration Management of Adaptive MPSoCs **69**
5.1 Motivation and Contribution . . . 70
5.2 Adaptive MPSoCs Implemented on FPGAs . . . 71
5.2.1 Hardware Architecture . . . 71
5.2.2 Application Software . . . 71
5.2.3 Task Implementation . . . 72
5.2.4 System Reconfiguration . . . 72
5.2.5 System Objectives . . . 73
5.2.6 High Level Modeling of Adaptive MPSoCs . . . 73
5.3 Modeling Reconfiguration Management Computation as a DCS Problem . . . 74
5.3.1 Architecture Behaviour . . . 74
5.3.2 Application Behaviour . . . 74
5.3.3 Task Execution Behaviour . . . 77
5.3.4 Global System Behaviour Model . . . 80
5.3.5 System Objectives . . . 81
5.4 Automatic Manager Generation by Using BZR . . . 81
5.4.1 BZR Encoding of the System Model . . . 82
5.4.2 Enforcing Logical Control Objectives . . . 83
5.4.3 Enforcing Optimal Control Objectives . . . 85
5.4.4 Enforcing the Combined Logical and Optimal Objectives . . . 87
5.5 Experimental Results . . . 87
5.6 Case Study . . . 89
5.6.1 Case Study Description . . . 89
5.6.2 Controller Generation and Integration . . . 90
5.6.3 System Implementation . . . 92
5.7 Summary and Discussion . . . 93

6 Combining Configuration and Reconfiguration Designs for Adaptive MPSoCs **95**
6.1 Motivation and Contribution . . . 95

6.2 A Design Flow for Adaptive MPSoCs 96
6.2.1 The Design Flow 96
6.2.2 Case Study: Continuous Multimedia Player 97
6.3 CLASSY for the Design and Evaluation of Run-Time Managers 105
6.3.1 An Illustrative Example 105
6.3.2 Run-Time Manager Design 106
6.3.3 Simulation and Analysis of the Designed Manager by using CLASSY 108
6.4 Summary and Discussion 111

7 Conclusion and Perspectives **113**
7.1 Conclusion 113
7.2 Perspectives 116

List of Figures **119**

List of Tables **120**

Bibliography **121**

Chapter 1

Introduction

Contents

1.1 Trends in Embedded Systems 1
1.2 Problem Statement 2
1.3 Contributions 3
1.4 Thesis Overview 5

1.1 Trends in Embedded Systems

Embedded systems are specific-purpose computer systems combining software and hardware components. They are used almost in everything that runs on electricity in our daily life. Examples include consumer electronics, avionics, telecommunication, automotive electronics, etc. Some trends can be observed in emerging embedded systems.

- First, the functionality integrated into embedded systems is ever increasing. Mobile phones, for example, have evolved from a simple device that supports only telephone calls over radio links to the so-called smart phones (e.g., Apple iPhone, Samsung Galaxy) that offer a wide variety of other services such as text messaging, Internet accessing, music and video playing, camera, gaming and photography. Another example is the car entertainment system [Schor *et al.* 2012], which integrates many similar functions such as GPS, music playing, radio etc.

- Second, the high integration of functionality into embedded systems leads to a tremendous increase in the amounts of data being processed by the systems. A mobile phone, for example, can contain gigabytes of video, photo and music data files to process. The amount of manipulated data is expected to double every two years in the future in these domains [Byna & Sun 2011]. To be made useful, the data must be processed in real-time for the users. The execution platforms must thus provide the required computational power to do this. On the other hand, power/energy consumption minimization becomes more and more important as many of these devices are battery powered. *Parallel execution platforms* play a key role for providing these applications with the required computational power to achieve data-intensive processing under real-time and energy-efficient constraints. In order to obtain adequate execution performances, a state-of-the-art solution consists in integrating multiple cores or processors on a single chip, leading to as **multiprocessor systems-on-chip** (MPSoCs) [Wolf *et al.* 2008]. For example, instead of accelerating the clock frequency of each new processor generation, Intel [Borkar *et al.* 2005] has shifted to a strategy for which multiple cores or processors are integrated on a single chip. Another example is the STMicroelectronics Nomadik [Artieri *et al.* 2003], which contains an ARM926EJ as its host processor, and two programmable accelerators for video and audio processing respectively.

- Third, *adaptivity*, the ability of a system to adapt itself, is becoming increasingly desirable in embedded systems. This trend can be observed in the following three aspects.

  – Embedded applications are becoming increasing adaptive. For example, a video-surveillance application for street observation needs to adapt its image analysis algorithms according to factors like the human activity (crowded place or not), luminosity (day or night) or the weather.

  – Hardware components are becoming increasingly adaptive. An example is the FPGAs, which allow run-time reconfiguration to implement different functionalities. It provides a better trade-off of performance and flexibility compared to application specific integrated circuits (ASICs) and general purpose processors. Another example is the *Dynamic Voltage/Frequency Scaling (DVFS)* technique, which allows the dynamic adjustments of supply voltage and clock frequency for hardware like processors so as to reduce power/energy consumption. A third example is the *clock gated* mechanism which allows processors or parts of circuits to switch from active mode to sleep mode so as to save power/energy.

  – Mapping and scheduling of applications are becoming increasing adaptive. For example, application tasks that run on a faulty or over-heated processor need to be dynamically migrated or remapped to another one.

  The *adaptation* ability allows a system to adapt its behaviors in reaction to changing environment conditions to maintain and/or optimize its behavior w.r.t. objectives on, e.g., safety, performance and power/energy consumption. However, it further complicates the system design.

- Fourth, the continuous increase in the size and complexity of future embedded systems, and the strict time-to-market pressures and design costs faced by the designers have demanded the use of *abstract models* for early design analysis, evaluation and validation. Traditional low-level approaches such as RTL level designs or physical prototyping on FPGAs (e.g., [Lee *et al.* 2006] [May *et al.* 2010]), are becoming too slow, tedious and even infeasible to meet design requirements. As a result, system level methodologies based on models at different abstraction levels (e.g., [Cai & Gajski 2003] [Thompson *et al.* 2007] [Stuijk *et al.* 2006b]) have been proposed to handle the given complexities with increased productivity and decreased time-to-market. Such methodologies perform design space exploration (DSE) and evaluation at an early design stage, and thus significantly reduce the design efforts. On the other hand, embedded systems are often safety critical, and must function correctly. Reliability and safety are more important than performance for such systems. *Formal modeling and analysis* techniques are thus required for their designs [Edwards *et al.* 1997].

1.2 Problem Statement

The trends outlined in the previous section show that with more and more new features integrated into embedded systems, multiprocessor systems-on-chips (MPSoCs) have become a state-of-the-art solution to achieve high performance and energy-efficiency. Their design complexity has significantly escalated. Meanwhile, they have to be adaptive, i.e., adapt their behavior regarding frequent environment changes for better execution performances, lower energy consumption, etc. Though the adaptivity feature can draw a lot of benefits,

it further complicates their design. This leads to a real challenge about cost-effective and safe design methodologies of adaptive/reconfigurable MPSoCs.

- Firstly, design correctness must be addressed to ensure system reliability in every possible system configuration.
- Secondly, reconfiguration correctness must also be established to safely control the variation between system configurations.

We further discuss these two design issues in some more details in the remainder of this section.

Modern embedded systems often execute multiple tasks concurrently, invoke and finish their executions at different moments in time. At different moments, different combinations of executing tasks form different *application configurations* or scenarios that an embedded system supports. A typical user expects that each application configuration provides predictable performance and reliability. This requires that the proper resource allocation and mapping (i.e., the binding and scheduling of application tasks on the allocated resources) decisions must be made for each application configuration w.r.t. system functional and non-functional requirements. We refer this design issue as **the design of a configuration of an adaptive MPSoC**.

Embedded systems are usually *reactive*, i.e., interact continuously with their environment at a speed imposed by the environment, and *adaptive*, i.e., must adapt their configurations in reaction to the run-time situations. To build such systems, the **reconfiguration management** issue, i.e., how to safely manage the system adaptive behavior w.r.t. system run-time situations and system requirements, must be addressed. Such a run-time manager needs to control and coordinate the system reconfiguration behavior in reaction to system run-time situations according to system requirements.

Especially, different run-time situations may require a different design decision for an application configuration. The reactive and adaptive features must be taken into account when addressing the first design issue, i.e., the design of a configuration of an adaptive MPSoC. The design for each application configuration thus should result in *a number of different solutions* that provide a trade-off in e.g., resource usage, performance and energy/power consumption, so that the run-time manager can choose among the design solutions w.r.t. run-time situations. This thesis aims to deal with these two identified design issues of adaptive MPSoCs.

1.3 Contributions

This thesis is carried out within the context of the French ANR project FAMOUS[1], the acronym for FAst Modeling and design flOw for dynamically reconfigUrable Systems. The FAMOUS project aims at introducing a complete methodology for the design of embedded systems, focusing on MPSoCs implemented on FPGA-based architectures. It covers research in the following three key design aspects of reconfigurable/adaptive embedded systems.

- High-level system modeling: it aims to define concepts necessary for modeling reconfigurable embedded systems through a Unified Modeling Language (UML) profile based on the Model-Driven Engineering (MDE) approach. The UML Modeling and Analysis of Real-Time Embedded systems (MARTE) profile, which provides a common way of modeling both hardware and software aspects of a system but does not

[1] http://www.lifl.fr/~meftali/famous/

contain any concepts for modeling dynamic and reconfigurable behaviors, is considered and will be extended to include concepts for modeling reconfigurable behaviors.

- Transformation and code generation: starting from high level application models, it aims to develop necessary model to model transformation rules, in order to allow safe and fast code generation for both simulation and synthesis.

- Verification and analysis: it aims to develop formal and automated verification and analysis methods for reconfigurable embedded systems, in order to guarantee that their final implementations correspond with the initial system requirements.

This thesis targets the *verification and analysis* task of the project. It makes several contributions to the identified design issues mentioned in the previous section.

- We propose a high-level framework named CLASSY for the rapid and cost-effective design space exploration (DSE) devoted to the design of adaptive MPSoCs. A multi-clock modeling of both software and hardware has been considered by exploiting the notion of abstract clocks borrowed from synchronous data-flow languages. Our approach is an ideal complement to lower-level design assessment techniques for MPSoCs, such as physical prototyping and simulation. It also aims to serve as an intermediate reasoning support that is usable, from very high-level MPSoC models (e.g., in UML MARTE profile), to deal with critical design decisions. The framework can be used to deal with the first design issue, i.e., the design of a configuration of adaptive MPSoCs. Furthermore, it is also flexible enough to capture the adaptive behaviors of MPSoCs, and can be used as a high level simulator for adaptive MPSoCs to evaluate customized run-time managers. Some results of this work are presented in [An *et al.* 2012a] [An *et al.* 2012b].

- We propose a general framework based on a tool-supported synchronous variant [Marchand & Samaan 2000] of the *discrete control* [Ramadge & Wonham 1989] for the reconfiguration management of adaptive MPSoCs. It favors automatic and correct-by-construction manager derivation. We illustrate our approach by considering MPSoCs implemented on FPGA-based reconfigurable architectures, which can draw various benefits such as efficiency and flexibility. In the framework, the system reconfiguration behavior is modeled in terms of synchronous parallel automata. The reconfiguration management computation problem w.r.t. multiple objectives regarding e.g., resource usages, performance and power consumption is encoded as a discrete controller synthesis (DCS) problem. The existing BZR programming language and Sigali tool are employed to perform DCS and generate a controller that satisfies the system requirements. The results of this work are presented in [An *et al.* 2013a] [An *et al.* 2013b] [An *et al.* 2013c].

- We investigate two different ways of combining the two proposed design frameworks for adaptive MPSoCs.

  – First, they are combined to construct a complete design flow for adaptive MPSoCs. The design flow starts from the MARTE high level system modeling, and then employs the two proposed design frameworks to respectively tackle the two design issues. At last, the design results are integrated into the original MARTE modeling framework, with which existing tools such as Gaspard2 [Gamatié *et al.* 2011] can be used to generate low level codes for further analysis and implementation.

– Second, they are combined to present how the CLASSY framework could serve as a high level simulator and assist the designers to design and evaluate run-time managers. To do this, the second framework based on the *discrete control* technique is used for designing a run-time manager, which is integrated into the CLASSY simulation process for analysis and evaluation.

1.4 Thesis Overview

This thesis is divided into two parts. Part I presents the state-of-the-art. It has two chapters. Chapter 2 presents the adaptive MPSoCs, discusses their design issues and existing design methodologies, and identifies the targeted design issues of the thesis. Chapter 3 presents the existing models, languages and tools that can be employed to address the targeted design issues. Among them, we identify and introduce the synchronous models, and discrete control techniques and tools that will be applied in the thesis.

Part II exhibits the contributions of the thesis. It has three chapters. Chapter 4 presents the abstract clock based design framework for the first design issue, i.e., the design of a configuration of an adaptive MPSoC. Chapter 5 presents the discrete control technique based design framework for the second design issue, i.e., the reconfiguration management of adaptive MPSoCs. Chapter 6 presents the two different ways of combining the two proposed design frameworks for adaptive MPSoCs. Finally, Chapter 7 concludes.

Part I

State of the Art

Chapter 2

Multiprocessor Systems-on-Chip (MPSoCs)

Contents

2.1 General Presentation 9
2.1.1 Processing Elements 10
2.1.2 Memory Architecture 11
2.1.3 Interconnect 12
2.2 MPSoC Design 13
2.2.1 A Design Flow Model 13
2.2.2 Design Space Exploration 15
Multi-Objective Optimization 15
Y-Chart Scheme for Automated DSE 16
2.2.3 A Survey of Existing Approaches 16
2.3 Adaptivity of MPSoCs 19
2.3.1 Motivation for Adaptivity in MPSoCs 19
2.3.2 FPGAs as Implementation Platforms for MPSoCs 19
2.3.3 Adaptivity Management 20
Some Existing Management Strategies 21
Run-Time Management of an FPGA Fabric 21
Autonomic Computing Approach 22
2.4 Summary and Discussion 23

An MPSoC is a system-on-chip that uses multiple processors as well as additional components such as peripheral devices and memories for an application. Nowadays, MPSoCs have emerged as the main solution for modern embedded systems [Siala & Saoud 2011], due to their ability to meet the computation requirements of their applications while keeping a low level power consumption. The adaptivity, an important feature of emerging MPSoCs, draws a lot of benefits for embedded systems, and meanwhile further complicates their design.

This chapter is organized as follows: Section 2.1 gives a general presentation of MPSoCs. Section 2.2 presents the design issues of MPSoCs. Section 2.3 focuses on the MPSoC adaptivity and its design issues.

2.1 General Presentation

The functionality integrated into embedded systems is ever increasing. This is typically observed in embedded multimedia systems. The Apple iPhone, for example, has a variety of applications that allow users to watch movies, listen to music, browse the Internet, send

emails, use online Google navigation, etc. The users typically expect that they have high performance, and at the same time, their energy consumption is kept at a minimal level, especially for battery-powered embedded systems. MPSoCs have been constructed to meet these requirements. In the following, we present the main components of MPSoC platforms.

The main components of an MPSoC platform are the processing elements (PEs), responsible for executing the applications, the on-chip memories, responsible for storing both the application data and instructions, I/O components, responsible for communicating with the outside world and, finally, the on-chip interconnect structure, responsible for linking the processing elements with the memories and the I/O components [Nollet 2008]. Figure 2.1 details the MPSoC platform template considered throughout this thesis. It is based on the tile-based multiprocessor platform described in [Culler *et al.* 1999] and consists of multiple tiles interconnected by an interconnect fabric. Each tile contains a local processing element, a local memory and a network interface (NI), which is accessed both by the local elements inside the tile and by the interconnect. Besides, the MPSoC platform could also contain some shared memory tiles to store large data sets (as shown in the third tile of Figure 2.1). In the rest of this section, we give some more details of the main MPSoC components.

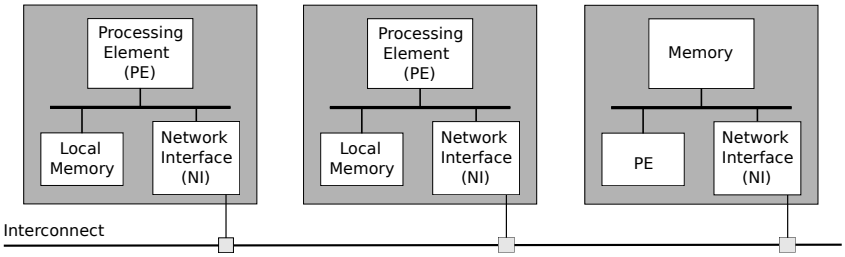


Figure 2.1: MPSoC platform template.

2.1.1 Processing Elements

Generally speaking, there exist three categories of processing elements: *application-specific integrated circuits (ASICs)*, *general-purpose processors (GPP)* and *reconfigurable logic*. These three computing technologies are quite different concerning their performance, energy efficiency and degree of flexibility. The ASICs are designed to perform some specific application, and can achieve the maximum energy efficiency and performance [Marwedel 2011]. However, they suffer long design times and the lack of flexibility. The key advantage of general purpose processors, on the contrary, is their flexibility and rapid developments: embedded system behavior can be changed by just changing the software running on such processors. However, they are usually very slow and much less energy-efficient. The reconfigurable computing technology combines the flexibility of general-purpose computing, and high performance and energy efficiency of application-specific computing.

The number and types of PEs contained in an MPSoC platform are obviously connected to the given application characteristics and requirements. Based on the types of PEs integrated in an MPSoC platform, two MPSoC architecture families can be distinguished: *homogeneous* and *heterogeneous* MPSoCs. In a homogeneous MPSoC, PEs are of the same type. Examples are multi- or many-core architectures used for general computing and PCs. Heterogeneous MPSoCs are composed of different types of PEs, such as micro-controllers, digital signal processing (DSP) processors, ASICs. In order to meet the computational performance of novel applications, while, at the same time, reducing

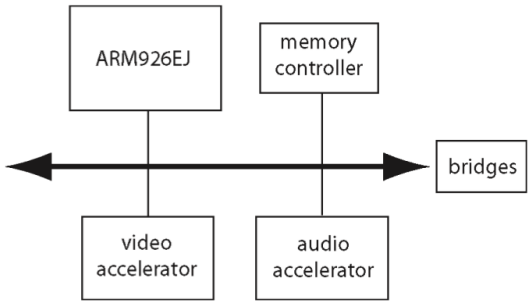


Figure 2.2: STMicroelectronics Nomadik platform, taken from [Wolf *et al.* 2008].

the power/energy consumption and remain flexible, there is a trend to use heterogeneous MPSoC platforms. In [Wolf *et al.* 2008], the authors present a wide range of MPSoC architectures developed over the past decade, such as STMicroelectronics Nomadik, Philips Viper Nexperia, Texas Instruments (TI) OMAP 5912. Figure 2.2, taken from [Wolf *et al.* 2008], shows the STMicroelectronics Nomadik [Artieri *et al.* 2003], an heterogeneous MPSoC platform for cellphones. It contains an ARM926EJ as its host processor, and two programmable accelerators for video and audio processing respectively. In the thesis, both architecture families are considered.

As mentioned at the beginning of this section, the key advantage of GPPs is their flexibility but they are energy inefficient compared to ASICs [Marwedel 2011]. To make them energy efficient, a number of techniques have been developed at various levels of abstraction [Burd & Brodersen 2002]. *Clock gating* is an example of such a technique. It is based on the consideration that clock signals do not perform any computation and are mainly used for synchronization, while they are a great source of power dissipation because of high frequency and load [Kathuria *et al.* 2011]. With clock gating, one can save power by disconnecting parts of a processor from the clock during idle periods [Marwedel 2011]. Another commonly-used technique that can be applied at a rather high abstraction level is the *dynamic voltage and frequency scaling* (DVFS). With DVFS, the clock frequency of a processor can be decreased at run-time to obtain a corresponding reduction in the supply voltage, which reduces power consumption and leads to significant reduction in the energy required for a computation [Le Sueur & Heiser 2010] [Milutinovic *et al.* 2009]. For example, the clock of the Crusoe^{TM} processor [Klaiber 2000] could be varied between 200 MHz and 700 MHz in increments of 33 MHz w.r.t. 32 voltages levels between 1.1 and 1.6 volts. It takes about 20 ms for the transition from one frequency/voltage pair to the next.

2.1.2 Memory Architecture

Memories are used to store data, programs, etc. Memory architecture has a direct impact on the performance and energy cost of MPSoCs. Memory access latency could make the PE computations to wait, and thus drop overall system performance. On the other hand, memory access also contributes to the overall system energy cost. As accessing smaller memories usually require less time and less energy, there is a trend to use new hierarchical memories organizations in MPSoCs instead of using a single on-chip memory. The hierarchy can store data used by a PE in a local memory close to it, such as the MPSoC template in Figure 2.1. Such a hierarchy can also avoid the access contention of PEs on a central shared memory.

Caches and scratch pad memories (SPMs) are examples of small memories. They both

have only one processor clock cycle access latency [Wolf *et al.* 2008]. Caches represent on chip interfaces between PEs and memories. It requires a hardware controller to check whether the cache has a valid copy of the data associated with a certain memory address. In the contrary, SPMs are mapped into the address space, and thus do not need a hardware controller. This makes SPMs more energy efficient than caches.

Some applications like multimedia applications may require access to quite large data sets. A video processing application, for example, might require access to a complete video-frame at HDTV resolution. Such a frame could be composed of over two million pixels, and each pixel needs three bytes of memory. In this case, the local small memories within a tile will typically not be large enough. An external memory is thus required in this case. This can be seen as a memory tile of the MPSoC platform template as shown in Figure 2.1.

2.1.3 Interconnect

There are a number of interconnect technologies that can be used by an MPSoC platform. The authors of [Siala & Saoud 2011] give a survey of existing interconnects w.r.t. communication topology and strategy. In the thesis, we mainly consider bus and Network-on-Chips (NoCs).

Bus is the traditional interconnection architecture in MPSoCs. The arbitration policy of the bus has a direct impact on the performances of an MPSoCs. The communication by bus has the main advantage of simplicity (i.e., a single channel of communication), and requires thus relatively less design time. On the other hand, the bus architecture is not efficient [Lee *et al.* 2008], since it has a limited bandwidth and the available throughput between two PEs connected to it is inversely proportional to the number of PEs connected to it [Siala & Saoud 2011]. Therefore, bus is the good choice for the architectures with small number of PEs, but it would cause low performance and high power/energy consumption issues when the number of PEs grows big.

Network-on-Chip (NoC) is an emerging paradigm for the communication within MPSoCs. An NoC consists of network adapters, routing nodes (or routers), and links that connect the routing nodes [Bjerregaard & Mahadevan 2006]. Routing nodes are used to route data according to their implemented routing strategies or algorithms. Network adapters are used to interface routing nodes with PEs. The authors of [Bjerregaard & Mahadevan 2006] characterize an NoC by its i) topology: nodes positions and connectivity, and ii) routing protocol: how the nodes and links are used for communication.

There exist a number of topologies for NoCs, such as Spidergon [Moadeli *et al.* 2007], Mesh [Ali *et al.* 2009], and Tree [Adriahantenaina *et al.* 2003]. We take the Mesh topology as an example to describe how such an NoC works. A $n \times n$ 2D Mesh topology NoC is formed by [Siala & Saoud 2011]:

- n*n routers, each router, except for those on the sides, is connected to 4 neighbor routers and a PE via input/output channels.
- an input/output channel performs unidirectional communication between two routers or between a router and a PE.

Figure 2.3 shows such a 4×4 Mesh topology NoC.

NoC routing protocols determine how a message packet traverses the NoC channels from its source router to its destination router [Chiu 2000]. Routing protocols can be deterministic or adaptive [Mirza-Aghatabar *et al.* 2007]. Deterministic protocols route packages by predefined paths [Chiu 2000]. An example is the "XY" routing algorithm for 2D meshed NoC presented above. It routes packages, from its source to its destination, first in the X

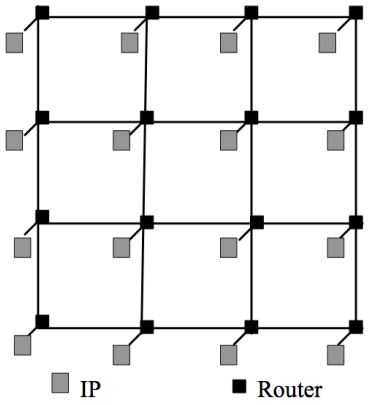


Figure 2.3: A 4×4 Mesh topology NoC [Ali *et al.* 2009].

direction then in the Y direction. Such protocols are deterministic, but lack flexibility and do not adapt to the network state dynamically. Adaptive protocols, on the contrary, detect the network traffic and channel status during the package routing, and adapt routing path so as to avoid congested regions of the network [Mirza-Aghatabar *et al.* 2007]. The routing protocol also plays an important role in the performance of MPSoCs, and the user thus needs to find out a proper one for his or her design.

Compared to bus, NoCs are more expensive in surface, but more scalable and efficient with less power consumption. NoCs provide a better compromise of cost and performance.

2.2 MPSoC Design

The design of modern embedded systems, which are usually based on MPSoC architectures, is becoming increasingly complex. To deal with this complexity, the common practice in this area is to raise the level of abstraction and adopt system level design methodologies [Gerstlauer *et al.* 2009] [Cannella *et al.* 2011]. Typically, system level design approaches follow a top-down approach. They rely on computational abstract models for the description of system functional and non-functional requirements, and traverse the design space by means of an iterative process, known as *design space exploration* (DSE), which evaluates and refines different design decisions to find an optimal solution. In this section, we firstly introduce the popular Y chart design flow model, which most of system level design approaches follow, in Section 2.2.1. Section 2.2.2 presents the design space exploration problem. A survey of existing design approaches is presented in Section 2.2.3.

2.2.1 A Design Flow Model

Embedded system design is a rather complex task, consisting of a number of sub-tasks such as functionality and platform modeling, application-architecture mapping, software code generation and hardware synthesis. A complete design flow must combine and perform these sub-tasks to generate the final systems. A very popular design flow model is the *Y chart*, as shown in Figure 2.4, proposed by Gajski and Kuhn [Gajski & Kuhn 1983]. In the Y chart, the design information is presented in three dimensions: behavioral, structural, and physical, represented by the three axes.

- Behavioral representation: it describes the system functionality, i.e., what the system does, and says nothing about the implementation and structure.

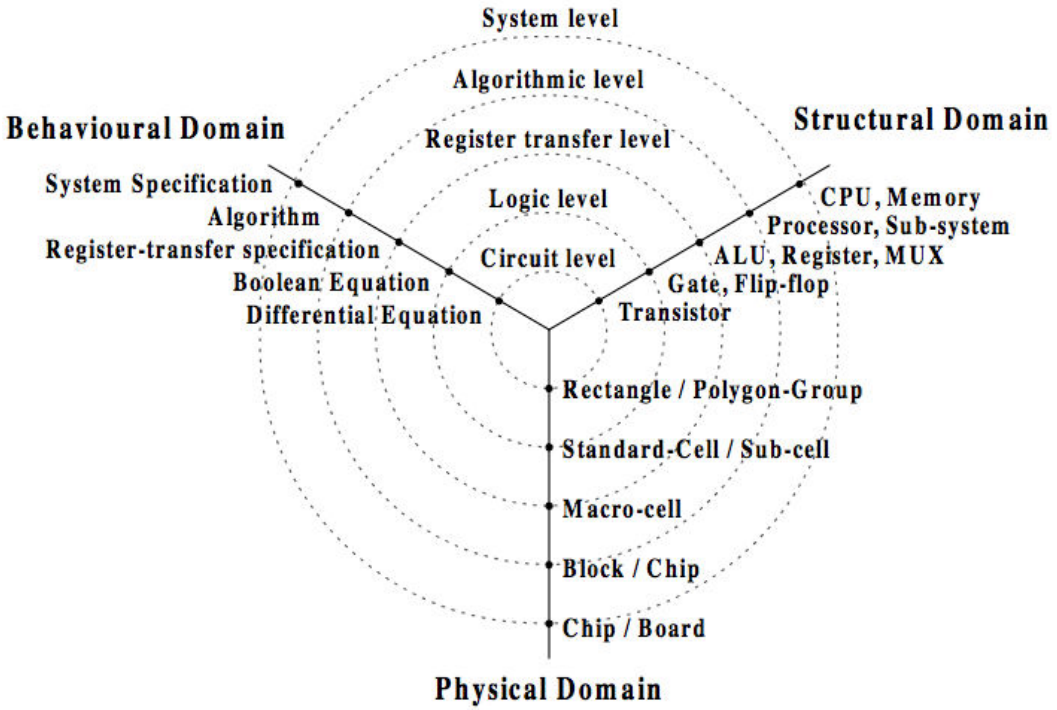


Figure 2.4: The Y chart design model, taken from [Tammemäe & Ellervee].

- Structural representation: it describes a set of computing components and connections that a behavioral representation maps onto. However, it does not specify any physical parameters, like the positions of components on a board.
- Physical representation: it is a layout planning, and describes the physical system.

Along the axes, the circles represent various levels of abstractions. Moving down along an axis represents moving down in the level of abstraction. In the graph, five levels of abstraction are identified: system level, algorithmic level, register transfer level (RTL), logic level and circuit level. Near the intersection points of the circles and the axes, the corresponding abstraction information corresponding the three dimensions are noted.

The considered level of abstraction has an great impact on the design analysis time and accuracy. Typically, the lower the abstraction level is considered, the more accurate analysis result can be obtained, while the more analysis time is required. This is due to the fact that more architectural details are taken into account when moving down in the level of abstraction. For example, the MPSoC design simulation at the register transfer level (RTL) could be about 400 times slower than at the transaction level (i.e., algorithmic level), as shown in [Boukhechem 2008]. However, RTL level is more accurate.

In the literature, nearly all system level design flows follow a top-down approach. They [Gerstlauer *et al.* 2009] typically start with a system level description consisting of a behavioral description, which is often some kind of data-flow graphs, and a platform model, which is typically a set of architectural components such as processors, memories. Additionally, some implementation constraints regarding mapping, performance, energy and cost are also given. A synthesis task is then performed to select an appropriate platform, determine a binding of the behavioral model to the platform, and generate an implementation, e.g., a scheduling on each platform resource. The resulting implementation is a refined model,

which integrates design decisions such as binding and scheduling. The refined model is then used as input to the design flow at lower levels of abstraction, where each software and hardware component is further implemented separately. In order to optimize the design for the synthesis task, a *design space exploration* (DSE) should be performed to evaluate and traverse the design space.

2.2.2 Design Space Exploration

The term "design space exploration" (DSE) has its origin in the logic synthesis context [Gries 2003]. Today, however, it usually deals with system-level design problems such as hardware/software partitioning, application-architecture mapping w.r.t. multiple objectives regarding, e.g., correctness, execution time, energy consumption. DSE is in general a multi-objective optimization problem. It is not advisable to merge all these objectives into a single objective function by e.g., using a weighted average, as this would hide some essential characteristics of designs [Marwedel 2011]. Returning a set of reasonable designs among which the designer can select an appropriate one is rather advisable. In this section, the multi-objective optimization is firstly presented. We then present the popular Y-chart scheme that enables systematic design space exploration.

Multi-Objective Optimization

Design Space Exploration (DSE) is a *multi-objective optimization problem* [Zitzler 1999] that tries to find out one or several "optimal" design solutions w.r.t multiple objectives regarding functional and non-functional properties such as correctness, execution time and power/energy consumption. The multi-objective design space exploration optimization problem can be defined as follows:

$$\min \text{ or } \max\ f(x) = (f_1(x), f_2(x), ..., f_n(x)) \text{ subject to } C.$$

where $x \in X$ is the decision vector, representing decisions such as which application tasks are mapped onto which PEs, C denotes the constraints (e.g., two tasks communicating with each other cannot be mapped onto two PEs that are not connected), and f is the objective function composed of n objectives $f_1, f_2, ..., f_n$.

Suppose $X_f \in X$ is the set of feasible solutions that meet constraints C. In single-objective optimization, the feasible set is totally ordered according to objective f: for any two solutions $a, b \in X_f$, either $f(a) \neq f(b)$ or $f(b) \neq f(a)$. The solution(s) optimizing objective f can always be found. However, when two or more objectives are involved, the feasible solution set X_f is, in general, not totally ordered any more. This situation is illustrated in Figure 2.5: the left gives a solution space that aims to maximize objectives $f1$ and $f2$, and the right describes the three possible relations of solutions. For any two solutions a and b, a **dominates** b if and only if a is better than b w.r.t. at least one objective and not worse than b w.r.t. all other objectives. For example, in Figure 2.5, A dominates B in the example where $f_1(A) > f_1(B) \wedge f_2(A) > f_2(B)$; C dominates D in the example where $f_1(C) > f_1(D) \wedge f_2(C) \geq f_2(D)$. A solution a is **indifferent** to solution b if neither a dominates b nor b dominates a. For example, E and B are indifferent, as they are better w.r.t. one objective and worse w.r.t. the other: $f_1(B) > f_1(E) \wedge f_2(E) > f_2(B)$. As a result, there is usually no single optimal solution for a multi-objective optimization problem, but rather a set of optimal trade-offs. A solution x is said to be **Pareto optimal** if and only if x is not dominated by any solution in X_f, e.g., the white points in Figure 2.5. The set of all Pareto optimal solutions form the **Pareto front** or **Pareto-optimal set**.

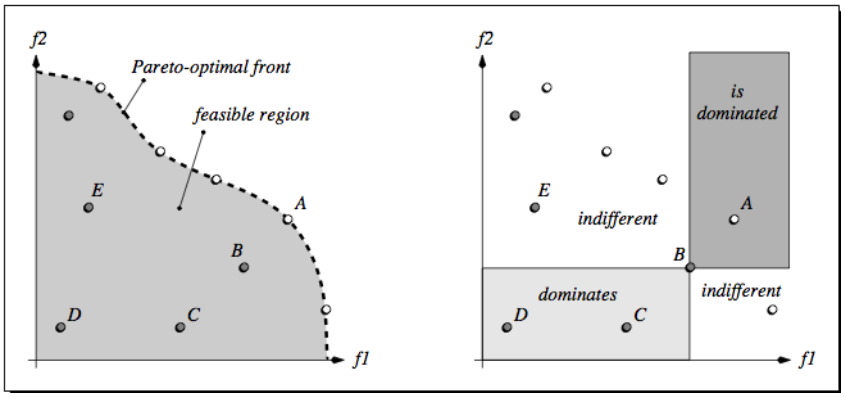


Figure 2.5: Illustrative example of Pareto optimality in objective space (left) and the possible relations of solutions in objective space (right), taken from [Zitzler 1999].

Design space exploration (DSE) based on Pareto points is the process of finding a set of Pareto-optimal solutions for the designer, enabling him or her to select the appropriate one(s) among them.

The solution space of the DSE problem becomes large quickly if arbitrary mapping is allowed. Considering an application consisting of m tasks and a platform with n PEs, the feasible mapping choices would be n^m. The complexity increases even further if multiple objectives are considered during the exploration. In order to evaluate if a design choice is Pareto-optimal w.r.t. a set of solutions, all objective values of the design must be compared with those of the other designs. The use of the exhaustive exploration of the design space is thus limited, if not infeasible. A common solution is to trade optimality for speed, and use heuristic techniques to guide the exploration process.

Y-Chart Scheme for Automated DSE

The *Y-chart scheme*, proposed by Kienhuis et al. [Kienhuis *et al.* 2002], presents a methodology that allows designers to perform systematic exploration of embedded system design space. It advocates *separation of concerns*, i.e., the separation of various design aspects to allow more effective design space exploration. Two fundamental separations are i) the separation of application behavior (what the system is supposed to do) and architecture (how it does it), and ii) the separation of communication and computation. Figure 2.6 shows the Y-chart scheme for design space exploration. It requires an explicit definition of the application and architecture models. The application model captures the application functional behaviors, whereas the architecture model describes the hardware resources as well as their performance constraints. A mapping step is applied to map application tasks onto architecture resources, and the performance analysis is then carried out to quantify design choices. This yields the performance numbers that designers interpret so that they can improve the designs by changing the design parameters such as the mapping denoted by dotted lines. This procedure is repeated in an iterative way until one or more satisfactory designs are found.

2.2.3 A Survey of Existing Approaches

As mentioned at the beginning of Section 2.2.1, embedded system design is a rather complex problem, and has a number of design tasks. A classification framework of design tasks has

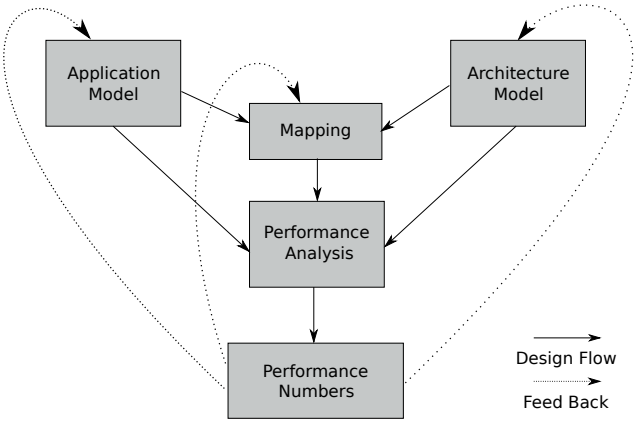


Figure 2.6: The Y-chart design scheme.

been defined in [Densmore & Passerone 2006], which identifies the individual design tasks, such as modeling, functional verification, software code generation, C-to-RTL synthesis, of more than 90 different design flows or methodologies. In contrary, [Gerstlauer *et al.* 2009] focuses on the complete design flows that combine all design tasks across hardware and software boundaries. In the thesis, we consider the system behavior and platform synthesis task, i.e., the application mapping design task as shown in Figure 2.6. *Application mapping*, which refers to the binding and scheduling of behavioral models onto platform models, has a strong impact on the quality of design results, and is the key design issue of MPSoCs [Marwedel *et al.* 2011].

Based on the Y-Chart scheme (see Figure 2.6), we survey and classify the existing approaches based on their used behavioral and architectural models. The behavioral model can be captured by model of computations (MoCs) [Lee & Sangiovanni-vincentelli 1998], programming languages like C, java, system level description languages such as SystemC, etc. We distinguish three levels of architecture modeling:

- system level models, which consists of an abstract architecture template, e.g., the PEs, memories, and interconnects, and for each architecture component, a list of supported behavioral model components and their performance requirements;
- transaction level model (TLM), which models the architecture by using system level description languages such as SystemC or SpecC;
- register transfer level (RTL) or lower level: which models architecture by using hardware description languages such as VHDL or Verilog.

The abstraction levels of architecture models have a great impact on the design analysis time and accuracy. In the following, some existing approaches are presented based on the identified architectural modeling levels.

We firstly introduce several complete design flows that allow gradual refinement of (abstract) system-level architecture models as mentioned in [Gerstlauer *et al.* 2009]. Most of the design flows cover all the identified architecture modeling levels.

- Daedalus [Thompson *et al.* 2007] provides a highly-automated framework for system-level architectural exploration and synthesis, programming, and prototyping of heterogeneous MPSoC platforms. It models the system behavior by means of the Kahn Process Network (KPN) MoC. It leads the designer in a number of refinement steps to

produce an MPSoC implementation on an FPGA at the RTL and ISA levels for hardware components and software processes, respectively. It uses a modeling and simulation environment called Sesame to perform system-level architectural DSE. Sesame supports both exhaustive and heuristic exploration methods.

- Metropolis [Balarin *et al.* 2003] provides a general framework which allows the description and refinement of a design at different levels of abstraction. The behavioral description is captured as a set of processes that communicate through channels. A backend is provided for the translation of meta-models into C++/SystemC simulation code.

- Koski [Kangas *et al.* 2006] provides a framework for modeling, automatic architectural design space exploration, and system level synthesis, programming, and prototyping of MPSoCs. The behaviral description is based on the KPN MoC from a UML description. It has an automatic architecture exploration step which transforms the application and architecture models to an abstracted model for fast exploration. It allows the automatic code generation to analyze and simulate the system at multiple levels of abstraction. The generated low-level software code and the RTL hardware descriptions (derived from its platform library) can be used for physical implementation.

- Ptolemy [Lee 2003] is an environment for simulation and prototyping of heterogeneous systems, which supports several different models of computation such as SDF, KPN, synchronous reactive models. It allows the designers to specify and simulate applications with different computational models on heterogeneous architectures at different levels of abstractions.

Some other examples of such design flows are PeaCE/HOPES [Ha *et al.* 2008], SystemCoDesigner [Keinert *et al.* 2009], etc.

Some other existing approaches are presented as follows following the identified architecture modeling levels from RTL to system level.

At RTL or lower level modeling of architectures, [Bailey & Martin 2010] applies physical prototyping which uses circuit board and SoC in the form of working silicon. The authors in [Hedde *et al.* 2009] propose a MPSoC prototyping platform that relies on field-programmable gate arrays (FPGAs) and register transfer level (RTL) descriptions. The major advantage of these two techniques is their high accuracy, but they require a long time and provide a limited flexibility when it comes to an efficient DSE of multiple architectures.

Design approaches that adopt the transaction level modeling (TLM) of architectures include [Petrot *et al.* 2011] [Abdi *et al.* 2011]. SystemC are usually used for the behavior modeling as well. Some simulation environments such as SoCLib [SoClib 2012] and StepNP [Paulin *et al.* 2002] use cycle-accurate model or TLM for hardware modeling and ISS for software simulation. The programming languages such as C, C++ are employed to describe the system behavior in these environments. Approaches based on MoCs and UML for behavioral modeling such as [Robert & Perrier 2010], [Chen *et al.* 2004] exist as well. The simulation speed and timing accuracy of TLM and ISS based techniques are faster, but less accurate than those of prototyping and emulation.

Since all these approaches based on TLM and RTL modeling of architectures need very detailed architecture information, they are very slow, tedious, and complex for cost-effective design and verification of modern MPSoCs that are complex and have big or even huge design space. They are thus only be used at the very late design stage.

At the early design stage, the system level modeling of architectures are usually considered. They usually use MoCs to model system behaviors. Some approaches based on Kahn Process Networks (KPNs) [Kahn 1974] are [Haid *et al.* 2009], [Schor *et al.* 2012]. Examples of approaches based on Synchronous Data Flow (SDF) [Lee & Messerschmitt 1987] are [Stuijk 2007], [Zhu *et al.* 2010], [Yang *et al.* 2009], [Stuijk *et al.* 2011]. The synchronous reactive MoC [Benveniste *et al.* 2003] is another well-known support for the high level modeling and analysis of embedded systems.

The design approaches based on system level modeling of hardware architectures abstract away the architectural information, and thus significantly reduce the design time. On the other hand, the missing of architectural information also makes the analysis results less accurate. Furthermore, their analysis results highly depend on the input performance model, i.e., the information of performance requirements of the application operations or tasks on resources. These values are usually obtained by statically profiling or analyzing corresponding worst case performance metrics. This could make the final results too pessimistic, and over-estimate the resource usages.

2.3 Adaptivity of MPSoCs

The adaptivity of an MPSoC refers to its ability to dynamically adapt its behaviors and structure over time. Reconfigurable hardware architectures (i.e., FPGA fabrics), due to their ability to combine some of the flexibility of software with the high performance of hardware, are becoming increasingly attractive for adaptive MPSoCs. In this section, the motivation for adaptivity is presented in Section 2.3.1. Section 2.3.2 introduces FPGAs as the implementation platforms for MPSoCs. Some adaptivity management issues are discussed in Section 2.3.3.

2.3.1 Motivation for Adaptivity in MPSoCs

Over the recent decades, there have been increasing requirements for embedded systems to be adaptive. The motivations for this trend can be observed in the following three levels:

- applications are becoming intrinsically dynamic: e.g., a surveillance embedded system for street observation must adapt its image analysis algorithms according to the luminosity of the weather.
- the execution platform should be adaptive to provide a better performance and/or reduce energy consumption: e.g., a hardware accelerator gives a more powerful execution in terms of performance than a general purpose processor; PEs that are idle should be turned off or switched to low level energy consumption mode, thanks to techniques like gate clocking.
- the mapping of the running application tasks should be adaptive to provide execution efficiency and/or fault tolerance: e.g., running application tasks need to adapt their mapping when some new tasks are invoked, so that they can execute on the shared computing resources in a more efficient way; a running task must adapt its mapping when some of its used resource becomes unavailable.

2.3.2 FPGAs as Implementation Platforms for MPSoCs

An Field-Programmable Gate Array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing. An FPGA is composed of an array of

logic cells and programmable routing channels to implement custom hardware functionality. The basic components of a logic cell (as shown in Figure 2.7) are a LUT (Look Up Table):

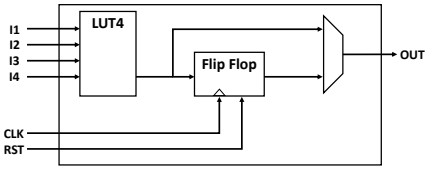


Figure 2.7: Simplified logic cell implementation.

a memory used as a programmable device to implement any logic function between inputs and outputs of a cell, and a D flip-flop: to hold a state between two clock cycles.

An FPGA configuration program consists of one or more *bitstream*s, which are binary files storing information to configure the LUTs and the routing switches. The bitstreams are usually generated by design tools such as the Xilinx Embedded Development Kit (EDK) [Xilinx 2013], which includes a tool suite called Xilinx Platform Studio (XPS) used to design an embedded system. Recent large FPGAs contain more than 200K logic cells that can be combined and interconnected to implement very complex designs. Multiprocessor architectures with tens of large hardware accelerators and processors can be implemented.

Run-time partial reconfiguration. In the new generation of FPGAs, the hardware configuration can be updated at run-time by using the partial reconfiguration feature. A portion or region of the FPGA which implements some logic functions can be swapped with another one. This feature also enables the FPGAs to update the functionality of any logic function at run-time if required. When multiple functions are called sequentially, the same region can be reused so that the required size can be minimised. The best advantage of this type of reconfiguration is its ability to reconfigure hardware during the running of the static part, i.e., the part which does not contain any reconfigurable area. It assumes that the hardware reconfiguration does not disturb the execution of the application. The bitstreams therefore cover only some regions of the FPGA array.

Such *Dynamically Partially Reconfigurable* (DPR) FPGAs make them suitable for addressing constraints on resources (re-using some areas for different functions for applications that can be partitioned into phases) by adapting resources to available parallelism according to environment variations. DPR FPGAs represent a trade-off in that they are slower than dedicated Application-Specific Integrated Circuit (ASIC) hardware, but much faster than software running on general purpose CPUs.

2.3.3 Adaptivity Management

The management of an adaptive MPSoC concerns three possible dynamic aspects (as mentioned in Section 2.3.1) of the system over time: dynamic application behavior, dynamic platform behavior and the dynamic application mapping. A run-time manager needs to monitor the system run-time situations, make adaptation decisions based on observed information and system requirements, and perform adaptation actions.

Reconfigurable hardware architectures (i.e., FPGA fabrics) have the ambition to deliver the same flexibility level as general purpose processors while providing a performance and energy efficiency level close to that of an ASIC. Such architectures are becoming increasingly attractive for MPSoCs. They operate in a very different way compared to a multi-core or instruction set processor (ISP) architecture. The run-time management of such specific architectures thus needs to be addressed differently.

The run-time management problem of adaptive systems can be seen as a self-management problem in *autonomic computing*. It proposes a general feedback loop structure address the automatic management of adaptive systems, which is well suited for the management of adaptive MPSoCs.

In the following, first, we present the existing management strategies for MPSoCs; second, the management issue of embedded systems implemented on FPGAs is discussed; at last, the autonomic computing paradigm is introduced.

Some Existing Management Strategies

Most of the existing approaches regarding the management of adaptive MPSoCs target the run-time mapping problem, i.e., how to map and schedule application tasks w.r.t. the availability of platform resources and system requirements. These approaches can be classified into two categories: purely run-time mapping approaches and hybrid design time/run-time mapping approaches.

Purely run-time mapping approaches do not have a pre-analysis phase. Finding a reasonably good solution in a reasonable short computation time, for such approaches, is more important than seeking optimality requiring much computation time. They thus resort to heuristic algorithms to generate fast and lightweight solutions on-line [Nollet *et al.* 2008]. For example, in [Smit *et al.* 2005] such a technique proposes an iterative hierarchical approach with simple heuristics in each individual level to solve the application mapping on a parallel heterogeneous SoC architecture at run-time. In [Ghaffari *et al.* 2007], an on-line partitioning algorithm is associated with a scheduling heuristic to address the application mapping problem. Since all the mapping analysis is done on-line, Such approaches can deal with the mapping of applications that are unknown a priori. However, such approaches cannot guarantee optimal solutions and/or strict system constraints, due to unknown situations, and are usually validated by (limited) simulations.

The hybrid approaches [Schor *et al.* 2012] [Schranzhofer *et al.* 2010] [Singh *et al.* 2011] firstly perform intensive analysis at design time, and then propose a run-time manager integrating the knowledge of design time analysis. However, they do not address run-time management problem systematically, and usually encode the run-time manager manually by considering a limited number of run-time configurations. This is tedious and error-prone [Gohringer *et al.* 2008] in consideration of more complex and dynamic system behavior.

Regarding the design verification, while simulation and testing are still widely considered in industry, they often imply a tedious and expensive validation process that necessarily requires a system implementation, some adequate test benches and a simulator to achieve experiments. Formal verification, such as model-checking, is therefore a useful complement to simulation and testing as it relies on abstract models [Mitra *et al.* 2010]. Two examples of applying model checking to validate the run-time manager design are [Adler *et al.* 2007] [Schaefer & Poetzsch-Heffter 2009], while the authors in [Yang *et al.* 2012] apply a game theory-based approach to construct a run-time manager.

Run-Time Management of an FPGA Fabric

The reconfiguration of a DPR FPGA fabric involves the process of loading configuration files to part of the reconfigurable surface. Figure 2.8 shows the structure of an FPGA chip, which is abstracted from the Xilinx ML605 board. The configuration files used for the different configurations of the partially reconfigurable regions are stored in a compact flash card. A soft-core processor e.g., microblaze is responsible for loading them. It performs the reconfiguration of the reconfigurable region through the Internal Configuration Access Port (ICAP).

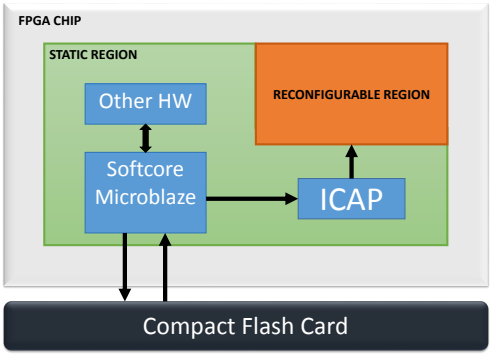


Figure 2.8: FPGA with a microblaze softcore.

The run-time management of reconfiguration involves a control loop, taking decision according to events monitored on the architecture, choosing the appropriate next configuration to install, and executing appropriate reconfiguration actions. The dynamism in the architecture dimension further increases the design complexity, for which a complete tool-chain is lacking [Santambrogio 2010].

Due to the relative novelty of DPR technologies, the management of reconfiguration has to be designed manually for important parts. For instance, [Quadri *et al.* 2010b] proposes a design flow, from high level models to automatic code generation, for the implementation of reconfigurable FPGA based SoCs. The system control aspects need to be modeled manually and integrated into the flow. Some other work employs model checking to verify the controller designs. For example, in [Dahmoune & Johnston 2010], the authors address post-silicon verification by connecting a model-checker to a physical implementation of reconfigurable systems on FPGAs.

In the current practice, though the numbers of considered tasks and reconfigurable regions are low (few units), ensuring a correct and optimal management by using manual encoding and analysis is tedious and error-prone [Gohringer *et al.* 2008]. Automatic techniques are required to better address this problem, with the foreseeable increase in complexity.

Autonomic Computing Approach

The autonomic computing paradigm [Parashar & Hariri 2005] [Kephart & Chess 2003], inspired by biological systems such as the autonomic human nervous system, enables the development of self-managing computing systems to handle the emerging complexity in computing systems, services and applications. The function of a self-management capability is a control loop that collects details from the system and acts accordingly [IBM 2006]. Typically, self-managing capabilities of autonomic systems can be classified into four categories:

- self-configuration: configure and reconfigure a system under varying conditions following high-level policies;
- self-optimization: detect sub-optimal behaviors and tune itself to optimize its execution;
- self-healing: discover, diagnose and recover from potential problems without disrupting the whole system environment;

- self-protection: detect hostile behaviors as they occur, protect itself from both internal and external attacks, and maintain system security and integrity.

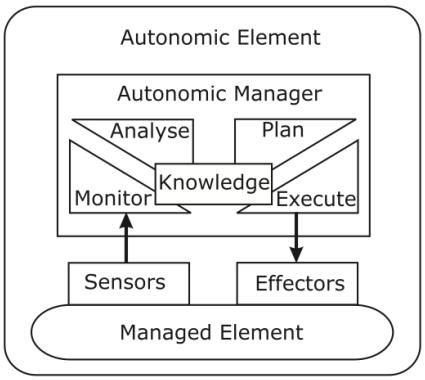


Figure 2.9: IBM's Monitor, Analyze, Plan, Execute, Knowledge (MAPE-K) reference model for autonomic control loops, taken from [Huebscher & McCann 2008].

To achieve autonomic management, IBM has suggested a reference model for autonomic control loops as shown in Figure 2.9, which is usually referred to as *Monitor, Analyse, Plan, Execute, Knowledge* (MAPE-K) [IBM 2006]. In the MAPE-K loop, the *managed element* represents any system software or hardware component that is managed by a coupled autonomic manager. To connect to the autonomic manager, the managed element requires a *sensor* to sense the changes in the internal and external environment, and an *effector* or actuator to react to and counter the effects of the changes in the environment. The autonomic manager consists of four components:

- the *monitor*, which collects the details from the managed resource via the sensor;
- the *analyze*, which models and analyzes the collected data by the monitor;
- the *plan*, which constructs the response actions needed to achieve objectives;
- the *execute*, which performs the execution of the plan via the effector.

All the four components function based on a *knowledge* representation of the system. The autonomic manager can be designed and developed in many different ways, relying on techniques e.g., heuristics, model checking, control theory, machine learning [Maggio *et al.* 2012].

2.4 Summary and Discussion

MPSoCs are becoming the main solution of modern embedded systems, as they can provide powerful computing ability at affordable power consumption. In this chapter, we firstly presented the main components of MPSoC platforms. Regarding the design of MPSoCs, which is quite complex and challenging, we have seen a common practice that advocates raising the levels of abstraction and adopting system level design methodologies. By reviewing the existing system-level design methodologies, and the design approaches at different levels of abstractions, we can observe the importance of performing design space exploration at early design stages. It can prune the design space when it is largest, and identify candidate design solutions in a fast and efficient way before performing lower level refinements for more accurate results, so that significantly reduce the design efforts.

Furthermore, MPSoCs are getting more and more adaptive. The adaptivity adds a new design dimension in addition to the design issue of MPSoCs as presented in Section 2.2 for which the adaptivity is usually not explicitly addressed. We have briefly reviewed the reconfiguration management strategies of adaptive MPSoCs, and focused on particularly adaptive MPSoCs implemented on reconfigurable architectures, i.e., FPGAs. Such architectures provide a good trade-off of flexibility and performance for implementing adaptive MPSoCs. Regarding the control of their reconfiguration, we have observed that manual encoding is adopted in most cases. In anticipating the increase in complexity, automatic manager derivation thus could be more favorable.

To sum up, we have looked at two design aspects of adaptive MPSoCs in this chapter: the design of MPSoCs without considering adaptivity, which can be seen as the design of a configuration of adaptive MPSoCs, and the reconfiguration management of adaptive MPSoCs. Embedded systems are typically reactive systems that are in continual interaction with their environment and run at a pace determined by that environment [Halbwachs 1993]. These two design issues correspond intrinsically to the analysis and control of reactive systems, for which there exist a variety of models, languages and tools.

Chapter 3

Models, Languages and Tools for Reactive Systems

Contents

3.1 Data-Flow Based Modeling Formalisms **26**
3.1.1 Kahn Process Network and Synchronous Data Flow 26
Kahn Process Network (KPN) 26
Synchronous Data Flow (SDF) 26
3.1.2 Data-Flow Synchronous Languages 27
Lustre 27
Signal 29
3.1.3 The UML Profile MARTE and CCSL Language 29
3.2 Automata-Based Modeling Formalisms **31**
3.2.1 StateCharts 31
3.2.2 Automata-Based Synchronous Languages 31
3.2.3 Heptagon Language 32
3.2.4 Formal Definition of Automata 33
3.3 Discrete Controller Synthesis (DCS) **35**
3.3.1 General Presentation 35
3.3.2 Control Objectives 36
Logical Control 36
Optimal Control 36
3.4 BZR Synchronous Language and DCS **37**
3.5 Some Approaches Applying Discrete Control **38**
3.6 Summary and Discussion **38**

Embedded systems are typically reactive systems, as they are in continual interaction with their environment and run at a pace determined by that environment [Halbwachs 1993]. In the last section, we identify the two design issues of adaptive MPSoCs in the thesis, namely the design analysis of one configuration and the control of reconfiguration. In this section, we will introduce the existing models, languages and tools for reactive systems that can be employed to address the two design issues.

Sections 3.1 and 3.2 present respectively the existing data-flow and automata-based modeling formalisms for reactive systems. In Section 3.3, we present discrete controller synthesis (DCS), a formal technique that can be applied on automata-based modeling formalisms for computing controllers to control the state transitions. The synchronous language BZR whose compilation process encapsulates the DCS operation is presented in Section 3.4. Some existing works that apply *discrete control* for computing systems are presented in Section 3.5.

3.1 Data-Flow Based Modeling Formalisms

Data flow models reflect the way in which data flows from component to component [Stephen A. Edwards 2001]. Each component transforms the data from one form to another. It is a very natural way of describing reactive systems.

A data flow model is specified by a directed graph, where the nodes or vertices represent computations and the links or edges represent communication channels [Marwedel 2011]. The computations of nodes are assumed to be functional, i.e.. based on the inputs only. The computation performed by each node is decomposed into a sequence of firings that are atomic actions. Each firing consumes and produces data tokens. Conceptually, each node performs its computation once its input data are ready. The only synchronization constraint is thus the dependent relations between data [Halbwachs 1993]. Since unrestricted data flow models are difficult to prove system properties, restricted models are commonly used [Marwedel 2011].

3.1.1 Kahn Process Network and Synchronous Data Flow

Kahn Process Network (KPN)

Kahn Process Networks (KPNs) [Kahn 1974] are a special case of data flow models. It consists of nodes and edges, where nodes correspond to computations performed by some tasks, and edges imply communications between nodes via channels that are one way infinite first-in, first-out (FIFO) queues with a single reader and writer. The computations are synchronized via a blocking read and non-blocking write protocol. KNP is deterministic in nature: for a given set of inputs, it will always generate the same outputs. KNP graphs show computations to be performed and their dependency, but not the total order in which computations must be performed. The computations of nodes are partially ordered.

Combining unlimited storage capacity in its unbounded FIFO buffers makes KPN a very expressive model [Geilen & Basten 2010]. Expressiveness of an MoC is usually in conflict with its analyzability, i.e., the ability to (statically) analyze a modeled application for its properties, such as deadlock-freedom, static scheduling, etc. KPNs, in general, need to be scheduled dynamically using run-time scheduling strategies such as first come first served algorithms, since it is difficult to predict their precise behavior over time [Marwedel 2011] due to their expressive power. Design approaches based on KNPs include Daedalus [Thompson *et al.* 2007], DOL [Thiele *et al.* 2007], Koski [Kangas *et al.* 2006], etc, as shown in Section 2.2.3 of Chapter 2.

Synchronous Data Flow (SDF)

Synchronous data flow (SDF) [Lee & Messerschmitt 1987] is another special case of data flow in which the amount of data tokens produced or consumed by each node on each links is specified a priori. These amounts are constant, and called rates. A node, which represents an atomic block of computation, can only fire or perform its computation if sufficient amount of input data are available on its incoming edges. The rates determine how often nodes have to fire w.r.t. each other such that an SDF graph can be executed in a repetitive pattern, called *an iteration*, that there is no net effect on the amounts of tokens in channels.

The restrictions imposed on the original data-flow architecture, i.e., atomic node computation and fixed rates, make the static scheduling and analysis of SDFs much easier. This is a significant advantage in comparison to KPN models. In [Lee & Messerschmitt 1987], the authors developed a whole theory to statically schedule SDF graphs on homogeneous

architectures. They proposed techniques for constructing periodic admissible sequential and parallel schedules, respectively referred to as PASS and PAPS. A period in PASS is constructed by computing the balance equations on data rates, while PAPS is achieved by constructing acyclic precedence graphs based on a number of periods of PASS. In the last years, analysis techniques for e.g., throughput [Ghamarian *et al.* 2006], buffer sizing [Stuijk *et al.* 2008] of SDF graphs have also been proposed. Such techniques abstract away the execution platforms, and provide useful insights for their implementations. These techniques have been implemented in the SDF3 tool-kit [Stuijk *et al.* 2006a].

SDFs are widely adopted for numerous design approaches as shown in Section 2.2.3 of Chapter 2. However, it has a couple of limitations. One main limitation is that SDFs cannot express conditional executions or dynamic applications.

3.1.2 Data-Flow Synchronous Languages

The synchronous approach [Benveniste *et al.* 2003] has been proposed in 80s to provide a rigorous mathematical semantics for the safe design of real-time, reactive systems. It relies on the *synchronous hypothesis*, which is a collection of assumptions [Potop-Butucaru *et al.* 2005]. The main assumption is made on *instants* and *reactions*. Behavioral activities are divided according to (logical, abstract) discrete time, which is a sequence of non-overlapping *instants*. Within each instant, input signals or variables possibly occur (for instance by being sampled), internal computations take place, and control and data are propagated until output values are computed and a new global system state is reached. This execution cycle is called the *reaction* of the system to the input signals. Based on the hypothesis, a number of synchronous languages, e.g., Esterel, Lustre, Signal, Argos, have been developed. This section focuses on the data-flow oriented synchronous languages, and among them introduces Lustre and Signal.

Lustre

Lustre [Halbwachs *et al.* 1991] is a declarative synchronous data-flow programming language for programming reactive systems as well as describing hardware. In Lustre, the system inputs and outputs are described by their flows of values along time. Time is discrete, and described as a sequence of instants. A flow takes its n-th value of the value sequence at the n-th instant of the time. Any Lustre program has a basic or reference clock, slower clocks can be defined by boolean-valued flows: a clock defined by a boolean-valued flow is the sequence of instants at which the flow has the value true.

Lustre operators. Classical operators over basic data types are available in Lustre:

- arithmetic operators: $+, -, /, \times, div, mod$;
- binary operators: and, or, not;
- conditional operators: $if\ then\ else$.

These operators are extended so as to process data flows that have the same clock. For example,

$$X = Y + 1 \text{ means that } \forall t \in N, X_t = Y_t + 1.$$

The conditional operator is different with regard to a classical one. For example,

$$X = if C > 0\ then\ Y\ else\ Z \text{ means that } \forall t \in N, if\ C_t > 0\ then\ X_t = Y_t\ else\ X_t = Z_t.$$

In addition to the classical operators, Lustre also provides temporal operators that can handle the value relations at different instants of the same clock or different clocks. We mention the following ones:

- *pre* is used to get the value of a variable at the previous instant, i.e., the value is memorized to be used at the next instant.
- -> operates on two operands owning the same clock. It is used to set the first value of a variable to the first value of another variable.
- *when* is called a sampling operator. It operates on two operands of the same clock. The second operand should be a Boolean variable. It samples the values of the first operand at the instants when the second operand is true.

Table 3.1 gives some examples of these operators.

t	t_1	t_2	t_3	t_4	t_5	t_6	t_7	...
X	1	2	3	4	5	6	7	...
$pre(X)$	$\perp$	1	2	3	4	5	6	...
3 -> X	3	2	3	4	5	6	7	...
Y	F	T	F	T	F	T	F	...
X *when* Y		2		4		6		...

Table 3.1: Examples of some Lustre temporal operators.

Lustre syntax. We give only a simplified version of Lustre syntax (see Figure 3.1), which is enough for the following presentation of the thesis.

```
node node_name ({input variable declaration list})
        returns ({output variable declaration list})
var {local variable declaration list}
let
       {set of equations}
tel
```

Figure 3.1: Skeleton of a Lustre node.

- Node: a Lustre programs, which fulfills a certain functionality. It is identified by its *node_name*. The node interface, i.e., input/output variables, are declared between corresponding brackets as shown in Figure 3.1. The program body which is composed of a set of equations is defined between keywords *let* and *tel*.
- Variable declaration: input/output variables as well as local variables, that are declared locally in a *var* statement, are declared in the form of *variable_name*: *type_name*.
- Equation: an equation is declared as an assignment $identifier = expression$;, where an expression is either a variable, or an operation on variables.

Signal

Signal [Guernic *et al.* 2002] [Gamatié 2010], which has been mainly developed in INRIA EP-ATR then Espresso team since 1980s, is a declarative, polychronous, data-flow synchronous language. It adopts a multi-clocked philosophy (thus is polychronous) to describe systems with multiple clocks, for which each component owns a local activation clock. Different to Lustre, which is a functional language, Signal is a relational language. The system behaviors are described using relations between the values of observed events, and the occurrences, also referred to as *abstract clocks*, of these events. The Polychrony toolset[1], developed based on Signal for embedded systems, provides a unified model-driven environment for the analysis, simulation, verification and synthesis of embedded systems from specifications to implementation.

3.1.3 The UML Profile MARTE and CCSL Language

The Unified Modeling Language (UML) [Object Management Group 2013b] is a graphical language for visualizing, specifying, constructing, and documenting information system. The main advantage of UML is that it provides a visual system expression to better understand the system, and benefit the communication between designers as well. The UML diagrams created by the designers can be used to generate code in various languages for design and analysis. However, being a general purpose modeling languages also make it too general. Specific semantics need to be added to UML in order to deal with the modeling and design of a specific domain.

The UML Profile for Modeling and Analysis of Real-Time and Embedded Systems named MARTE [Object Management Group 2013a] is a standard proposal of OMG to support the specification, design, and verification/validation of real-time and embedded system. It is intended to replace the existing UML Profile for Schedulability, Performance and Time.

Among the rich set of concepts that MARTE offers, we mention the following ones that can be used to describe different features of embedded systems. The *General Component Modeling* (GCM) package is used to define general aspects such as algorithms in the application software part of a system. The *Hardware Resource Modeling* (HRM) package is used to describe hardware architecture, e.g., processors and memories. The *Allocation* package serves to define software/hardware mapping. Furthermore, for data-intensive applications such as image or video processing, data-parallel algorithms and multiprocessor execution platforms are described with the *Repetitive Structure Modeling* (RSM) package.

CCSL (Clock Constraint Specification Language) [André & Mallet 2008] [Mallet 2011] is a declarative companion language, inspired by several time models of the concurrency theory, of the specification of the UML MARTE Profile. CCSL is based on the notion of clock which represents a set of discrete event occurrences, called instants. A clock can be either chronometric or logical. Chronometric clocks are a means to model "physical time" and to measure duration between two instants. Logical clocks represent discrete time composed of abstract instants called ticks. The number of ticks between two instants may have no relation to any "physical duration". CCSL can be used to associate abstract clocks with UML components such as ports. Then, the interaction between components via the events occurring on their ports can be characterized by abstract clock relations. All these mentioned packages are useful in the description of each system configuration.

Reconfiguration modeling formalism will be detailed in the next section. Concerning reconfiguration modeling in MARTE, package *Configurations* can be used to describe different configurations, scenarios, or modes, of a system, while UML *Finite State Machines*

[1] http://www.irisa.fr/espresso/Polychrony

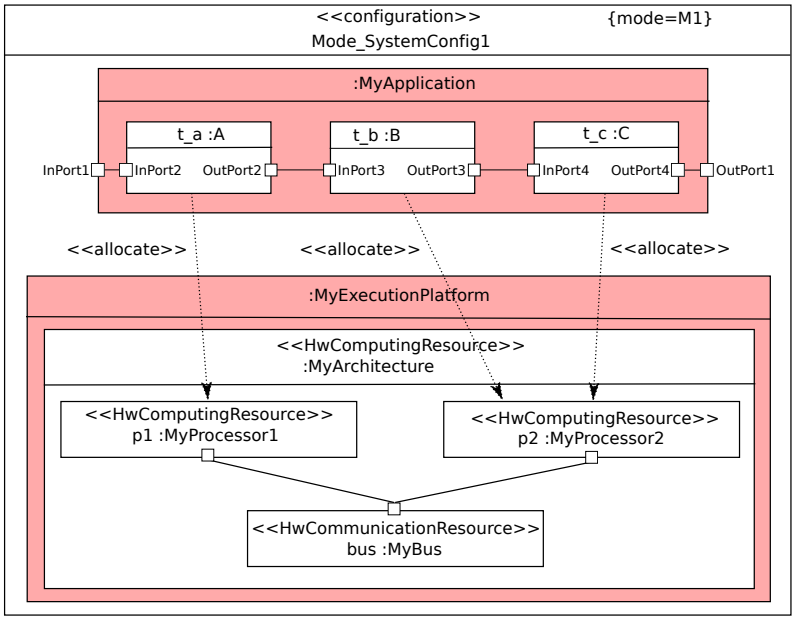


Figure 3.2: A system configuration modeled by MARTE.

can be used to describe configuration switches.

Figure 3.2 illustrates a UML Component representing a system mode or configuration named $Mode\_SystemConfig1$, with the stereotype $<< configuration >>$. A UML *Component* is used to represent a modular part of a system. There is no restriction on the granularity of a component, and a larger system part can be assembled by reusing components as parts in an encompassing component. Here, the configuration component is such an encompassing component. It encapsulates the details of the configuration, which is defined as an allocation or mapping of an application, consisting of three tasks A, B and C, on an hardware architecture, consisting of two processors connected by a bus. The application and its tasks, the architecture and its resources are all defined as components. A component may need to interact with other components or environment. The UML Ports, depicted as small squares on the sides of components are used to specify the interaction points between components. A connection between two ports represents that there exists some interaction, e.g., data transformation, between the two associated components. <<>> enclosing a name represents a stereotype defined in some package. For example, stereotype $<< allocate >>$ besides a dotted arrow is defined in the *Allocation* package. It represents an allocation of an application task on a hardware resource. $<< HwComputingResource >>$ and $<< HwCommunicationResource >>$ are defined in the *Hardware Resource Modeling* (HRM) package. The mode $Mode\_SystemConfig1$ component also contains a *mode* attribute, which has value $M1$, noted on the top right.

The way mode values are produced for selecting configurations is specified via an UML FSM, as modeled in Figure 3.3 with stereotype $<< modeBehavior >>$. The FSM contains two states corresponding to two configurations. Mode values defined in configuration components (e.g., M1 in Figure 3.2) are associated with the corresponding FSM states. The transitions between states is captured via stereotype $<< modeTransition >>$ between the FSM states.

A hardware/software co-design framework called Gaspard2 [Gamatié *et al.* 2011] dedicated to high-performance embedded systems has been proposed to enable the designers

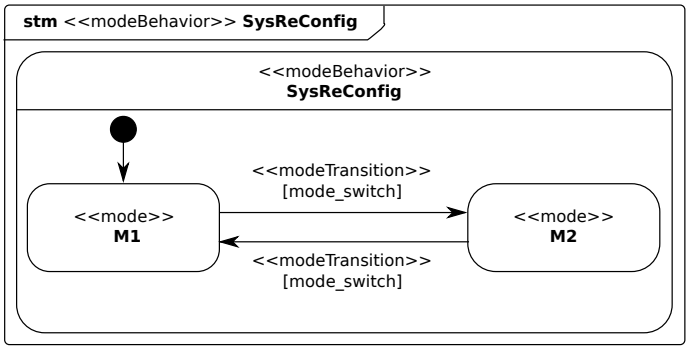


Figure 3.3: A system reconfiguration behavior modeled by MARTE.

to automatically generate low level codes, e.g., SystemC, synchronous languages such as Signal, VHDL, from high level MARTE descriptions for simulation, formal verification and hardware synthesis.

3.2 Automata-Based Modeling Formalisms

3.2.1 StateCharts

StateCharts [Harel 1987] is probably the first formal language designed for the reactive system design. It is a very prominent example of a language based on automata, and supports modular, hierarchical descriptions of system behavior, catering for multi-level descriptions as well as concurrency [Drusinsky & Harel 1989]. Its semantics has been defined at a sufficient level of detail in [Drusinsky & Harel 1989]. Another advantage is that a number of commercial tools based on StateCharts exist, such as StateMate and StateFlow. Most of them support the translation of StateCharts into equivalent descriptions in C or VHDL.

StateCharts has many features of synchronous languages such as synchronous product and broadcast mechanism, and is the origin of some automata-based synchronous languages such as Argos [Maraninchi & Rémond 2001]. A variation of StateCharts, called *state machine diagram* is also included in UML, though their semantics are not completely the same.

3.2.2 Automata-Based Synchronous Languages

Argos [Maraninchi & Rémond 2001] is a synchronous language based on parallel and hierarchic automata. It takes its origin in StateCharts, but solves some existing problems such as those concerning modularity and causality loops [Halbwachs 1993]. Its semantics is formalized and compatible with the synchronous point of view adopted in Esterel [Halbwachs 1993].

SyncCharts [André 1996], created by Prof. Charles Andre, is another example of automaton based synchronous languages. It inherits many features from StateCharts and Argos. SyncCharts consists of states and transitions for its structure, and signals for its dynamics. Any SyncCharts program can be automatically translated into an Esterel program, which can benefit the users from the software environment developed for synchronous programming.

Mode-Automata [Maraninchi & Rémond 2003] are a new programming construct that helps building systems that exhibit clear “running modes”. They support the programming

of mixed Argos-style automata with Statecharts-like parallel and hierarchical composition and Lustre-style data-flow equations labeled on the states of automata.

Another example is the Heptagon synchronous language, inspired by Mode-Automata, [Colaço *et al.* 2005], etc. It is considered in the thesis, and introduced with some more details in the next section. Some data-flow synchronous languages have also been extended with automata to deal with control-oriented designs, e.g., Signal [Brunette *et al.* 2009].

3.2.3 Heptagon Language

Heptagon language programs behave as synchronous Moore machines, with parallel and hierarchical composition. It supports the programming of mixed synchronous data-flow equations (like Lustre, as shown in Section 3.1.2) and automata with parallel and hierarchical composition.

Similar to Lustre, Heptagon structures the programs in *nodes* for scalability and abstraction purpose. A node has a name, a set of input flows, a set of output flows and the body defining the component behavior. Its basic behavior is that: at each step, according to inputs and current state values, equations associated to the current state produce outputs, and conditions on transitions are evaluated in order to determine the state for the next step. Inside the nodes, this is expressed as a set of equations defining, for each output and local, the value of the flow, in terms of an expression on other flows, possibly using local flows and state values from past steps.

The control structure, i.e., automata, is inspired from the Mode-Automata presented in [Maraninchi & Rémond 2003]. Figure 3.4 illustrates an example of the control behavior

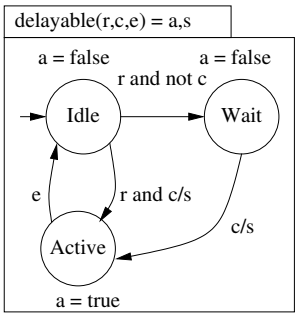


Figure 3.4: The graphical description of a delayable task.

of a delayable task. It is structured in a *node*, with name *delayable*, input flows r, c, e and output flows a, s. It describes three task states: idle, waiting and active. When it is in the initial Idle state, the occurrence of the `true` value on input `r` *requests* the starting of the task. Another input `c` can either allow the activation, or temporarily block the request and make the automaton go to a waiting state. Input `e` notifies termination. The outputs represent, respectively, `a`: activity of the task, and `s`: triggering starting operation in the system's API.

The graphical description of Figure 3.4 can be encoded in Heptagon as shown in Figure 3.5. The keywords to encode automata are shown in italics. Especially, Heptagon supports two types of transitions: *weak and strong transitions*.

- weak transition, encoded by *until* c *then* S, indicating that when c becomes true, the current state is executed before leaving it for target state S. The target state S is only executed at the next instant.

- strong transition, encoded by *unless c then S*, indicating that when c becomes true, the automaton instantly leaves the current state, and the target state S is executed.

```
node delayable(r,c,e:bool) returns (a,s:bool)
 let
        automaton
        state Idle
           do a = false; s = r and c;
           until r and c  then Active
           | r and not c  then Wait
        state Wait
           do a = false; s = c;
           until c  then Active
        state Active
           do a = true; s=false;
           until e  then Idle
        end
 tel
```

Figure 3.5: The textual description of a delayable task.

The nodes can be reused by instantiation, and composed in parallel or in a hierarchical way. Figure 3.6 shows an example with two instances of the *delayable* node put in parallel (noted by ";") defined in the *twotasks* node. When nodes run in parallel, one global step corresponds to one local step for every node.

The Heptagon compilation produces executable code in target languages such as C or Java. They have the form of an initialization function *reset*, and a *step* function implementing the transition function of the resulting automaton. It takes incoming values of input flows gathered in the environment, computes the next state on internal variables, and returns values for the output flows. It is called at relevant instants from the infrastructure where the controller is used.

```
node twotasks(r1,e1,r2,e2:bool) returns (a1,s1,a2,s2:bool)
  let
          (a1, s1) = delayable(r1, c1, e1);
          (a2, s2) = delayable(r2, c2, e2);
  tel
```

Figure 3.6: The textual description of two delayable tasks put in parallel.

3.2.4 Formal Definition of Automata

In the thesis, we adopt the formal framework defined in details elsewhere [Altisen *et al.* 2003] [Dumitrescu *et al.* 2010] for the automata definition. Besides, automata and Labelled Transition Systems (LTS's) are used interchangeably in the thesis.

Definition 1 (Automaton). *An automaton is a tuple* $S =< \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} >$, *where:*

- $\mathcal{Q}$ *is a finite set of states;*

- $q_0 \in \mathcal{Q}$ *is the initial state of* S*;*
- $\mathcal{I}$ *is a finite set of input events;*
- $\mathcal{O}$ *is a finite set of output events;*
- $\mathcal{T}$ *is the transition relation that is a subset of* $\mathcal{Q} \times Bool(\mathcal{I}) \times \mathcal{O}^* \times \mathcal{Q}$*, such that* $Bool(\mathcal{I})$ *is the set of Boolean expressions of* $\mathcal{I}$ *and* $\mathcal{O}^*$ *is the power set of* $\mathcal{O}$*.*

Each transition, denoted by $q \xrightarrow{g/a} q'$, has a *label* of the form g/a, where *guard* $g \in Bool(\mathcal{I})$ must be true for the transition to be taken, and *action* $a \in \mathcal{O}^*$ is a conjunction of output events, emitted when the transition is taken. State q is the *source* of the transition, and state q' is the *destination*. A *path* is a sequence of transitions denoted by $p = q_i \xrightarrow{g_i/a_i} q_{i+1} \xrightarrow{g_{i+1}/a_{i+1}} ... \xrightarrow{g_{i+k-1}/a_{i+k-1}} q_{i+k}$, where $\forall j, i \leq j \leq i+k-1, \exists (q_j, g_j, a_j, q_{j+1}) \in \mathcal{T}$.

The composition of two automata put in parallel is the *synchronous composition*, denoted by $||$. Given two automata $S_i =< \mathcal{Q}_i, q_{i,0}, \mathcal{I}_i, \mathcal{O}_i, \mathcal{T}_i >, i = 1, 2$, with $\mathcal{Q}_1 \cap \mathcal{Q}_2 = \emptyset$, their *composition* is defined as follows: $S_1||S_2 =< \mathcal{Q}_1 \times \mathcal{Q}_2, (q_{1,0}, q_{2,0}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T} >$, where $\mathcal{T} = \{(q_1, q_2) \xrightarrow{g_1 \wedge g_2 / a_1 \wedge a_2} (q_1', q_2') | q_1 \xrightarrow{g_1/a_1} q_1' \in \mathcal{T}_1, q_2 \xrightarrow{g_2/a_2} q_2' \in \mathcal{T}_2, g_1 \wedge g_2 \wedge a_1 \wedge a_2\}$. Composed state (q_1, q_2) is called a *macro state*, where q_1 and q_2 are its two *component states*.

The *encapsulation* operation, defined in [Altisen *et al.* 2003], is used to enforce the synchronization between two composed automata by means of a variable which is an input on one side, and an output on the other side. Let $S =< \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} >$ be an automaton, and $\Gamma \subseteq \mathcal{I} \cup \mathcal{O}$ be a set of inputs and outputs of S. The *encapsulation* of S w.r.t. Γ is the automaton $S \backslash \Gamma =< \mathcal{Q}, q_0, \mathcal{I} \backslash \Gamma, \mathcal{O} \backslash \Gamma, \mathcal{T}' >$ where $\mathcal{T}'$ is defined by $(q \xrightarrow{g/a} q' \in \mathcal{T}) \bigwedge (g^+ \bigcap \Gamma \subseteq \mathcal{O}) \bigwedge (g^- \bigcap \Gamma \bigcap \mathcal{O} = \emptyset) \Leftrightarrow (q, \exists \Gamma . g, \mathcal{O} \backslash \Gamma, q') \in \mathcal{T}'$. g^+ is the set of variables that appear as positive elements in the monomial g, i.e., $g^+ = \{x \in g | (x \bigwedge g) = g\}$. g^- is the set of variables that appear as negative elements in the monomial g, i.e., $g^- = \{x \in g | \neg (x \bigwedge g) = g\}$. Figure 3.2.4 gives an example of using encapsulation to enforce the synchronization of two automata A and B that are composed by a synchronous composition through variable b.

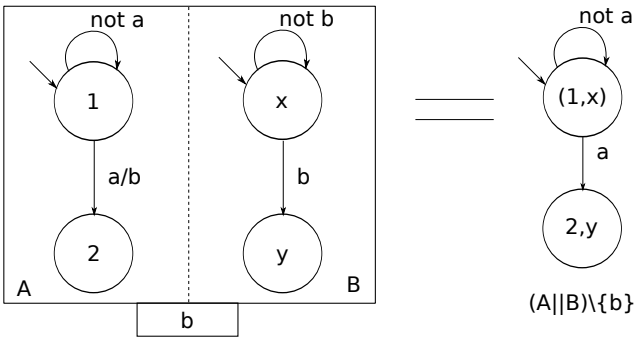


Figure 3.7: An example using encapsulation to enforce the synchronization of two composed automata.

The states of LTS's can be associated with *weights*, characterising corresponding quantitative features. We define a cost function $C : \mathcal{Q} \rightarrow N$ to map each state of an LTS to a positive integer cost value. Costs can also be defined on execution paths across an LTS. For instance, a cost function of path p can be defined as the sum of all the costs of its traversed states. When composing LTS's, the cost values w.r.t. the resulting global states/transitions can be defined on the basis of the local costs as their sum or the maximal/minimal value.

Based on the above definition of automata, and other automata-based modeling formalisms presented in this section, formal analysis and verification techniques, such as model checking, discrete controller synthesis, can be applied. In the thesis, we adopt the *discrete controller synthesis* technique, which is presented in the next section.

3.3 Discrete Controller Synthesis (DCS)

3.3.1 General Presentation

Discrete controller synthesis (DCS), introduced in 1980s by [Ramadge & Wonham 1989], was proposed to deal with the control and coordination problems of discrete event systems. A *discrete event system* (DES) [Ramadge & Wonham 1989] is a discrete-state, event-driven dynamic system that evolves in accordance with the occurrences of discrete events at possibly unknown irregular intervals. An event, for example, may correspond to the invoke or completion of a task, the failure or frequency switch of a processor. Such systems arise in various domains of our daily life, such as manufacturing, transport, automotive, embedded systems, healthcare. These applications have their own design requirements, and require control and coordination to ensure their desired behavior.

The main advantage of the theory is that it separates the concept of open loop dynamics (i.e., the DES) from feedback control, and allows the autonomic analysis and control of DESs w.r.t. a given specification of control objectives.

Discrete controller synthesis (DCS) is an operation that applies on a DES presented as e.g., a labeled transition system as defined in 3.2.4. In order to control a DES, i.e., enable a controller to influence the evolution of the DES behavior, it is postulated that the occurrences of certain events are under control. The set of inputs Y of a DES is thus partitioned into two subsets: Y_{uc} and Y_c, representing respectively the *uncontrollable* and *controllable* event sets. Figure 3.8 shows the principle of discrete controller synthesis (DCS). It is applied with a given control *objective*: a property that has to be enforced by control. The objective is expressed in terms of the system's outputs X. The controller denoted by C is obtained automatically from a system model S and an objective, both specified by a user, via appropriate synthesis algorithms. The synthesis algorithms, which are related to model checking techniques, automatically compute, by exploring the system state space, a constraint on controllable variables Y_c, i.e., the controller. Its purpose is to constrain the values of controllable variables Y_c, in function of outputs X and uncontrollable inputs Y_{uc}, such that all remaining behaviors satisfy the given objective.

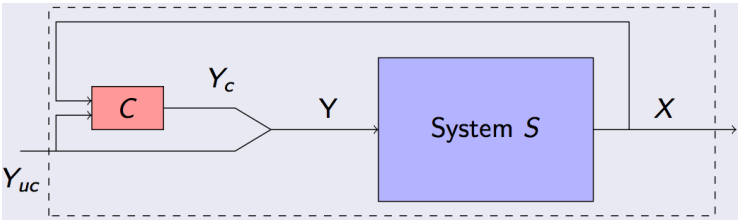


Figure 3.8: Principle of discrete controller synthesis

There can be several controllers that meet the same control objective. In the extreme case, a controller can forbid any state transition in order to avoid the invalid states. This is apparently not desirable for target systems. We are interested in *maximally permissive* controllers, which ensure the largest possible set of correct behaviors of the original uncontrolled system.

3.3.2 Control Objectives

This section presents the control objectives considered in the thesis. They are defined in terms of states and transitions of a LTS-modeled discrete event system. We categorize them into logical and optimal ones. The synthesis algorithms corresponding to these objectives exist in the literature, and have been implemented in the tool Sigali[2].

Logical Control

The following two logical control objectives [Girault & Rutten 2009] are considered:

- invariance of a subset of states E: a function $S' = make\_invariant(S, E)$ that synthesizes and returns a controllable system S' such that the controllable transitions leading to states $q_{i+1} \notin E$ are inhibited, as well as those leading to states from where a sequence of uncontrollable transitions can lead to such states $q_{i+k} \notin E$.

- reachability of a subset of states E: a function $S' = keep\_reachable(S, E)$ that synthesizes and returns a controlled system S' such that the controllable transitions entering subsets of states from where E is not reachable are disabled. Note that making E invariant is equivalent to making states not in E unreachable.

Algorithms corresponding to these two objectives can be found in [Marchand & Samaan 2000]. They have been implemented in Sigali as controller synthesis operations *S_Security(S,prop)* and $S\_Reachable(S, prop)$ respectively, where S denotes the system model and *prop* represents the target subset of states.

Optimal Control

Costs or weights can be defined on the states and/or transitions of a LTS as described in Section 3.2.4. Optimal control objectives specify how to control the transitions in order to optimize some costs. We distinguish one-step optimal and optimal control on path objectives.

One-step optimal control. One-step optimal control objective is to minimize/maximize, in one step, some function w.r.t. the costs or weights associated with states or transitions, i.e., to control the system go only to the next states with optimal weight or trigger only transitions with optimal weights. There can be several equally weighted solutions, so optimization does not necessarily lead to determinism. It can be noted that this gives us only a one step choice i.e., a local optimal, not a global optimal on all the behaviors.

Algorithms for optimizing and minimizing a cost function in one step can be found in [Marchand & Samaan 2000].They have been implemented by Sigali operations $Strictly\_Greater\_than(S, C, C\_Dup, duplicate\_states)$ for maximizing a cost function, and $Strictly\_Lower\_than(S, C, C\_Dup, duplicate\_states)$ for minimizing a cost function, where

- S denotes the system model,

- C denotes the cost function, associating states with corresponding costs,

- C_Dup denotes the duplicated cost function, and

- *duplicate_states* denotes the duplicated states of S.

[2] http://www.irisa.fr/vertecs/Logiciels/sigali.html

Optimal control on path. The optimal-control-on-path objective is to optimize the costs accumulated along paths of bounded length across a LTS. A path of length k starting at state q_1 and ending at state q_k is the sequence of $k-1$ transitions between them. The cost of a path is defined as the sum of the costs of all the states and/or transitions along the path. An optimal control algorithm that drives a LTS from the current state towards the target states at the best cost despite the worst moves of uncontrollable events can be found in [Dumitrescu *et al.* 2010]. This has been implemented by the Sigali operation $S\_min\_weight\_path\_maxUC(S, C, T)$, where

- S denotes the system model,
- C denotes the cost function, associating states with corresponding costs, and
- T denotes the set of target states.

Note that this operation is not available in the current version of the Sigali tool.

3.4 BZR Synchronous Language and DCS

BZR[3] extends Heptagon with a new behavioural *contract* [Delaval *et al.* 2010], and encapsulates the DCS tool Sigali [Marchand & Samaan 2000] in its compilation process. The *contract* enables users to specify the control objectives to be enforced in a declarative style. The system behavior is described, in an imperative style, in terms of automata by the Heptagon synchronous language (see Section 3.2.3). The compilation of BZR will automatically synthesize a controller enforcing the specified objectives. This controller is then re-injected automatically into the initial BZR program so that an executable program can be generated (in C or Java) for execution.

```
twotasks(r1, e1, r2, e2)
      = a1, s1, a2, s2
enforce not (a1 and a2)
with c1, c2

  (a1, s1) = delayable(r1, c1, e1) ;
  (a2, s2) = delayable(r2, c2, e2)
```

Figure 3.9: A BZR program with contract.

Figure 3.9 shows an example of a BZR program with a contract coordinating two instances of the `delayable` task defined in Figure 3.4 of Section 3.2.3. The `twotasks` node has a `with` part that declares controllable variables c_1 and c_2, and the `enforce` part that asserts the objective to be enforced by the DCS. Here, we want to ensure that the two tasks running in parallel will not be both active at the same time, defined by `not` (a1 `and` a2). Thus, c_1 and c_2 will be used by the computed controller to block some requests, leading the automaton of a task to the waiting state whenever the other task is active.

The controller or constraint produced by DCS is maximally permissive as discussed in the end of Section 3.3.1. This implies that several solutions might be valid for the values of controllable variables. To generate deterministic executable code for simulations, the BZR compiler chooses the solution by giving priority, for each controllable variable, to value `true` over `false` when both values are valid, and evaluating the controllable variables following the order of declaration in the `with` statement.

[3] http://bzr.inria.fr

3.5 Some Approaches Applying Discrete Control

The application of discrete control methods and techniques to computing systems is only emerging, although the classical control theory has been readily applied to computing systems [Hellerstein *et al.* 2004].

In [Iordache & Antsaklis 2009], discrete control is applied for the synthesis of concurrent programs based on Petri net models, while the authors of [Liu *et al.* 2008] deals with the schedulability analysis of Petri nets. Some other works have focused on the deadlock avoidance problem in shared memory multi-threaded programs by applying discrete control. In [Wang *et al.* 2009], the authors propose a programming language-level approach, that relies upon Petri net models, to synthesize deadlock-avoidance control logic for deadlock avoidance. [Auer *et al.* 2009] [Wang *et al.*] are some of the other works that also apply discrete control to deadlock avoidance. [Gaudin & Nixon 2012] handles another software problem, which applies discrete control to modify programs in such a way that the run-time exceptions that cannot be handled by the code will be inhibited.

In an approach related to reactive systems and synchronous programming, discrete controller synthesis, that defined and implemented in the tool Sigali, has been integrated in the compiler of BZR (see Section 3.4). [Delaval *et al.* 2010] describes how the BZR compilation works to perform invariance control, with modular DCS computations. BZR has been applied in the work concerning component-based software [Bouhadiba *et al.* 2011], in the work concerning the coordination control of administration loops [Gueye *et al.* 2012], and in the work concerning reconfigurations of reconfigurable architectures [Guillet *et al.* 2012]. Compared to [Guillet *et al.* 2012], more elaborate DCS algorithms would be applied, and the integration into a design flow and compilation chain would be more developed, in the thesis. Other works related to synchronous programming concern the articulation between reactive programs and DCS [Marchand *et al.* 2000] [Altisen *et al.* 2003], the manual application of the DCS to embedded systems [Gamatié *et al.* 2009], and the application of DCS to fault-tolerance [Girault & Rutten 2009] [Dumitrescu *et al.* 2010].

3.6 Summary and Discussion

This chapter presented the existing models, languages and tools for embedded systems, which are reactive systems. The data-flow based models, which are very natural to describe system behaviors, are firstly presented. KPN, SDF and data-flow synchronous languages such as Lustre and Signal are all high level models that are able to capture the system behavior of embedded systems. They are very suitable to deal with the first design issue, i.e., *the design of a configuration of adaptive MPSoCs*, identified in Chapter 2. The data-flow synchronous languages, in particular, have the concept of abstract clocks, which make them suitable to model the hardware platform (contains multiple PEs running at their own clocks) as well. Moreover, they have formal mathematical semantics, which can enable formal analysis. This is very important for embedded systems, which are usually safety-critical.

W.r.t. the second design issue identified in Chapter 2, i.e., *the reconfiguration management of adaptive MPSoCs*, the automata based modeling formalisms are presented. They are quite natural to model system reconfiguration behaviors. Among the modeling formalisms, we adopt the automata-based synchronous languages to address this design issue, as they have a rich set of formal analysis techniques. To design a safe controller for the reconfiguration management of MPSoCs, we choose the discrete controller synthesis technique. Compared to model checking, which gives indications or diagnosis on the original

design when a bug is detected and requires the designer to go back to the design and modify it before performing the verification again, DCS is more constructive as it is able to synthesis a controller directly. The BZR synchronous language that encapsulate the DCS in its compilation is then presented. At last, we present some existing works that apply discrete controller synthesis for computing systems. It is observed that discrete control is rarely applied to the reconfiguration management of reconfigurable architectures, i.e., FPGAs.

Based on the models, languages and tools presented in this chapter, the following two chapters will present our contributions to the two identified design issues respectively.

Part II

Contributions

Chapter 4

CLASSY: A High Level Design and Analysis Framework for MPSoCs

Contents

4.1 Motivation and Contribution **44**
4.2 High-Level Modeling of Adaptive MPSoCs **45**
4.3 An Abstract Design Framework for Adaptive Systems **47**
4.3.1 Application Behavior 47
4.3.2 Execution Platform Behavior 48
4.3.3 Application Mapping on Platforms 49
4.4 Scheduling and Design Analysis **49**
4.4.1 Clock Modeling of System Executions 49
4.4.2 Admissible Scheduling of Applications 50
4.4.3 Scheduling Algorithm 51
4.4.4 Performance Analysis 54
4.5 Design Space Exploration **55**
4.6 Implementation and Experiments **57**
4.6.1 CLASSY Tool 57
4.6.2 Experimental results 58
*Exp.*1 : Analysis precision w.r.t. low level cycle-accurate simulation environment 59
*Exp.*2 : High level design analysis 61
*Exp.*3 : Scalability 64
*Exp.*4 : Supporting system design 64
*Exp.*5 : Dealing with dynamic behavior 65
4.7 Summary and Discussion **66**

This chapter presents our contribution to the first design issue, i.e., the design of a configuration of adaptive MPSoCs. Our aim is to provide a fast and cost-effective means to assist the designers to make early implementation (i.e., mapping and platform configuration) decisions. In consideration of the system adaptivity, we target a number of Pareto-optimal implementation candidates, which provide a trade-off in resource usage, performance and energy/power consumption. They can be used by a run-time mechanism, presented in the next chapter, to adapt the mapping in different run-time situations.

The chapter is organized as follows: Section 4.1 exposes the motivations and contributions of the study. Section 4.2 introduces the input specifications of system applications and execution platforms. Section 4.3 presents our abstract modeling and analysis framework.

Section 4.4 addresses application scheduling on MPSoCs and performance evaluation. Section 4.5 presents our implementation and reports experimental results. Finally, Section 4.7 concludes.

4.1 Motivation and Contribution

Embedded systems are special-purpose computer systems combining software and hardware components that are subject to external constraints coming from environment and execution platforms. Their implementation on chips, referred to as systems-on-chip (SoCs) makes them pervasive, ubiquitous and suitable in many modern applications. Examples are consumer electronics among which, the most emblematic products are mobile smart-phones.

With the high *integration* of functions, embedded systems have become very smart and sophisticated. Nowadays, smart phones provide users with a large panel of facilities for communication, music and video players, built-in camera, Internet access, etc. Future embedded multimedia systems are expected to keep on integrating more functions.

All these facilities within a single system lead to the processing of huge amounts of information. A mobile phone can contain gigabytes of video, photo and music data files to process. Many modern embedded application domains include *data-intensive processing*: they operate on large data sets or streams, where the processing mostly consists of data read/write and data manipulation.

Moreover, embedded applications often have real-time constraints. In video processing applications, there are usually rate and deadline constraints imposed to image display. When executed in mobile devices, further concerns such as power/energy consumption become very important for battery life. *Parallel execution platforms* play a key role for providing these applications with the required computational power to achieve data-intensive processing under real-time and energy-efficient constraints. In order to obtain adequate execution performances, a state-of-the-art solution consists in integrating multiple cores or processors on a single chip, leading to as *multiprocessor systems-on-chip* (MPSoCs) [Wolf *et al.* 2008].

Adaptivity is another desirable feature in embedded systems. First, the ability to adapt to environment variations becomes very important. For instance, a video-surveillance embedded system for street observation adapts its image analysis algorithms according to factors like the human activity (crowded place or not), luminosity (day or night) or the weather. Some video-processing systems need to adapt their data processing according to the consumption and production rates of input and output dataflows.

Our contribution. We consider a high-level design and analysis framework for MPSoCs w.r.t. the trends mentioned previously. Traditional development approaches would program embedded data-intensive applications using data parallel languages or programming models such as OpenMP or message passing interface (MPI) according to the shared or distributed nature of memory in target platforms. We believe this abstraction level for programming MPSoC-based applications is tedious because of the error-prone manual coding and debugging efforts for programmers, particularly with the increasing complexity of modern embedded systems. In this context, we advocate the elevation of design abstraction levels for parallel embedded systems in order to favor faster and cost-effective design approaches, while permitting a relevant analysis of complex design spaces. Here, we propose a modeling and analysis framework for the design space exploration of adaptive applications on MPSoCs. An application is represented according to its event occurrences with their precedence relations. We analyze various design scenarios of its mapping and scheduling on an MPSoC by using abstract models. Among properties of interest are behavioral correct-

ness, execution times and energy consumption. A major advantage of our approach is that it offers a simple and fast alternative to explore and reduce complex design spaces before applying alternative but more expensive techniques, such as physical prototyping and low-level simulations. Our design framework is seen as an intermediate design-assistance tool usable between high-level MPSoC-based models and their implementations. Some results in this chapter have been presented in [An *et al.* 2012a] [An *et al.* 2012b].

4.2 High-Level Modeling of Adaptive MPSoCs

The starting point for our approach is a specification comprising a high-level description of executed applications and the associated execution platforms. Such a specification is typically modeled with the UML Marte profile [Object Management Group 2013a]. Here, we statically define the different scenarios (or modes) that characterize possible system behaviors. Let us consider a motion JPEG (M-JPEG) decoding application example. The adaptive behavior of the M-JPEG application is decomposed into two configurations as illustrated in Figure 4.1 and 4.2 in terms of application component graphs. The former captures the initialization phase while the latter represents the nominal pipelined execution. Beyond these examples of configurations, note that it is possible to capture via the same abstract specification concepts, the fact that the functionality of a given component changes from one configuration to another, or a variation of its input/output data size according to different configurations, etc.

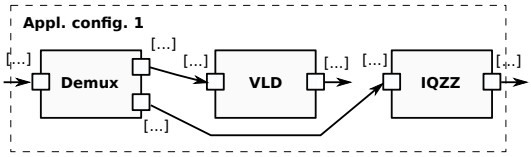


Figure 4.1: An initial application configuration.

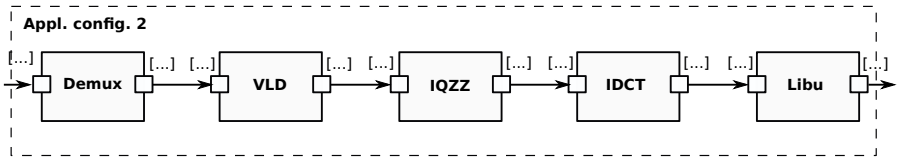


Figure 4.2: A nominal application configuration.

The way the specified configurations switch between each other during execution is described in a manager, typically defined as a finite state machine. An example is depicted in Fig. 4.3, where each state corresponds to a configuration. The cyclic transition from state *AppConfig2* denotes the multiple successive execution of the nominal mode (it consists of a periodic execution performed 36 times).

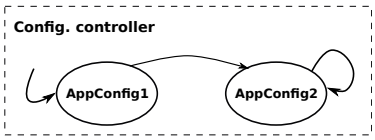


Figure 4.3: Application configuration manager.

The execution platform of the M-JPEG decoder can be specified as shown in Figure 4.4. It consists of a set of processing elements interconnected by a communication infrastructure that can be typically a Network-on-Chip (NoC). At this abstract level, the only required information about communications is, for each direct path linking two different processing elements, the best-case and worst-case latencies. On the other hand, since processors are supposed to run at variable frequency levels, the possible values for these frequencies are also provided in the specification. Our design space exploration framework therefore investigates which choices in these frequency values are suitable w.r.t. functionality correctness, temporal performance and energy consumption.

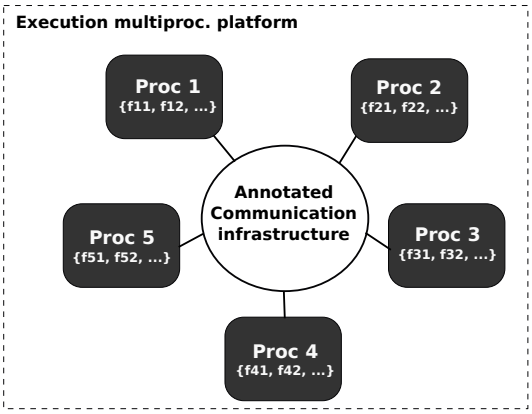


Figure 4.4: A multiprocessor platform model.

As for applications, note that configurations could be also considered for execution platforms. For instance, one may consider configurations with distinct architecture topologies, with different types of processing elements (e.g., processor versus FPGA), with different frequency values of processing elements, etc. Finally, our reasoning framework requires pre-profiled information for each elementary component of an application, regarding its execution latency and energy consumption on processing elements to be considered in the design space. Such an information should be also provided in the form of best-case and worst-case latencies.

The above abstract specification can be satisfactorily modeled with high-level modeling formalisms such as MARTE [Object Management Group 2013a]. This profile defines a rich set of concepts to describe different features of adaptive embedded systems. The *General Component Modeling* (GCM) package is used to define general aspects such as algorithms in the application software part of a system. The *Hardware Resource Modeling* (HRM) package is used to describe hardware architecture, e.g. processors and memories. The *Allocation* package serves to define software/hardware mapping. The CCSL package is used to associate abstract clocks with UML components such as ports. Then, the interaction between components via the events occurring on their ports can be characterized by abstract clock relations. All these packages are useful in the description of each system configuration.

Concerning reconfiguration modeling, we also need additional features: *Configurations*, which is used to describe different implementation scenarios, or modes, of a system, and UML *Finite State Machines*, which is used to describe configuration switches.

The Gaspard2 hardware/software codesign framework [Gamatié *et al.* 2011] dedicated to high-performance embedded systems adopts Marte as input design model. It allows one to automatically generate from such a model, simulation code in Pthreads, OpenCL or OpenMP Fortran for massively parallel systems. The approach we present in this paper

can be conveniently integrated in such a framework before code generation, so as to help a design to rapidly explore his/her design space and identify the most efficient system design choices.

4.3 An Abstract Design Framework for Adaptive Systems

We present how the input high-level system models mentioned in Section 4.2 are addressed based on our abstract design and reasoning framework.

4.3.1 Application Behavior

We first consider a *static application behavior*, i.e., an *application configuration*, defined via an application component graph. Tasks exchange data according to the connections specified in the graph. Each task has its own local activation clock according to which an associated sequence of events is observed. We use the tagged signal system presented in [Lee & Sangiovanni-vincentelli 1998] to construct our models.

The following sets are assumed: a discrete set $\mathbb{T}$ of logical instants, having a smallest element τ_{min} and associated with a partial order $\leq$; and a value domain $\mathbb{V}$. Then, let us consider the following definition of static application behavior:

Definition 2 (static application behavior)**.** *Given an application composed of a set T of tasks, its static behavior is a pair $(\mathcal{E}, \prec)$ with $\mathcal{E}$ the set of all possible events associated with all tasks, i.e., $\mathcal{E} = \bigcup \mathcal{B}_t, t \in T$ and a precedence relation $\prec$ defined on events of $\mathcal{E}$.*

The behavior $\mathcal{B}_t$ of task t is defined as a sequence of events, and an event e is a pair (τ, v), where $\tau \in \mathbb{T}$ is a logical instant, and $v \in \mathbb{V}$ is a value.

We extend the previous definition to define *dynamic (i.e., adaptive) application behaviors*. The dynamic behavior of considered embedded applications may depend on: *i)* the nature of data to be processed, e.g., different behaviors of the MPEG 4 decoder [Stuijk *et al.* 2010] for I frames and P frames; *ii)* environmental conditions, e.g., corresponding to different behaviors of smart cameras for good and bad weather situations [Wildermann *et al.* 2010]; and *iii)* execution platform resource availability. Such an application can be captured by a set of static behaviors in combination with a *controller* managing its dynamism by taking into account all relevant factors. In our vision, a dynamic application behavior therefore consists of a sequence of static behaviors over time, under the control of an associated manager or controller.

A dynamic application, with a task set T, has a set of static behaviors. We call it the *behavior set* of the application, denoted by $\mathcal{B}_T$. Given a controller $\mathcal{C}(\mathcal{B}_T)$ that determines its static behaviors over time, the dynamic application behavior is defined as follows:

Definition 3 (dynamic application behavior)**.** *Given an application composed of a set T of tasks, with a behavior set $\mathcal{B}_T$ and a controller $\mathcal{C}(\mathcal{B}_T)$, its dynamic behavior is a (possibly infinite) sequence of static behaviors denoted by a tuple $(\mathcal{C}(\mathcal{B}_T), \omega)$, where $\omega \in \mathbb{N}^*$ is the number of application iterations over time, $\mathcal{C}(\mathcal{B}_T)$ is a controller determining a behavior from $\mathcal{B}_T$ for each step (or iteration) of the sequence.*

Figure 4.5 shows the set $\{b_1, b_2\}$ of static behaviors for an application. Suppose it has a dynamic behavior alternating $\{b_1, b_2\}$ for 20 iterations, then its dynamic behavior is denoted by $(\mathcal{C}(\{b_1, b_2\}), 20)$ where $\mathcal{C}(\{b_1, b_2\}) = b_i, i = ((j-1) mod\ 2) + 1$, s.t. j is an iteration index from 1 to 20.

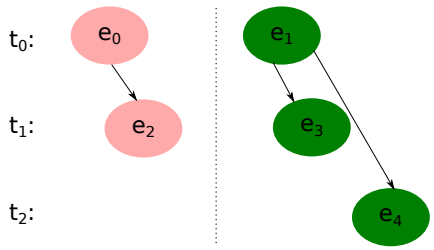


Figure 4.5: Set of two behaviors b_1 and b_2.

The application behavior definitions given previously are quite abstract. They have abstracted away the communication concerns, and the precedence relations between events can be interpreted as data transmission or any general controlled sequence. This allows the designer to focus on some application analysis like functional behavioral correctness without considering irrelevant aspects.

The communication aspect, however, plays an important role in embedded system design and analysis. Given the input specifications of Section 4.2, the set of communicating channels, and the data production and consumption sizes to and from channels should then be extracted to capture the application communication aspect.

In our framework, given a behavior $(\mathcal{E}, \prec)$ and a set of channels Ch in an application specification, we define a function $\delta : \mathcal{E} \times Ch \rightarrow N$ to capture the data production and consumption operations of events w.r.t. channels. N is an integer representing the operating data size. $N > 0$ and $N < 0$ indicate respectively a data production and consumption operation, while $N = 0$ means no data exchange between the corresponding event and channel.

4.3.2 Execution Platform Behavior

We consider heterogeneous execution platforms consisting of a set P of processing elements (PEs) communicating via an interconnect $\mathcal{I}$, e.g., a network-on-chip. Each PE is assumed to be associated with a local memory with enough storage space. The interconnect is characterized by the *best-case and worst-case transmission bandwidths*, e.g., bytes/millisecond, for all pair of PEs. A selection strategy is defined to choose values within this interval during simulation. Each PE $p_i \in P$ has a set of possible operating frequencies denoted by $fs(p_i) = \{f_i^j\}, j = 1, 2, ...$, and can adapt its frequency during execution, i.e., dynamic frequency scaling (DFS). The need to adapt the platform configuration or behavior (i.e., the operating frequencies of PEs) usually comes from the energy efficiency [Chen & Kuo 2007], which aims to minimize energy consumption while all tasks are done in time. Such frequency adaptations usually come with overheads that cannot be ignored. We define a combination of the frequencies of all PEs as a *static platform behavior*, i.e., a *platform configuration*, denoted by $b_P = \{f_i, 1 \leq i \leq |P|, f_i \in fs(p_i)\}$. All the PE frequency combinations thus comprise the *behavior set* $\mathcal{B}_P$ of the platform. Similar to application controllers, a platform controller $\mathcal{C}(\mathcal{B}_P)$ is in charge of platform behaviors over the time.

We model a platform behavior through the clock activations of PEs by considering the inverse of their frequency values, i.e., clock cycles. A reference clock $\mathcal{K}$ is defined to synchronize the activations of PEs. The frequency $f_\mathcal{K}$ value of the reference clock $\mathcal{K}$ is calculated as the least common multiple (LCM) of frequencies of all PEs: $LCM(fs(p_1), ..., fs(p_{|P|}))$. A cycle $1/f_i^j$ of a PE p_i is thus equal to an integer number of cycle $1/f_\mathcal{K}$. Figure 4.6 illustrates a dynamic behavior of a platform composed of three PEs p_0, p_1 and p_2 with frequency sets

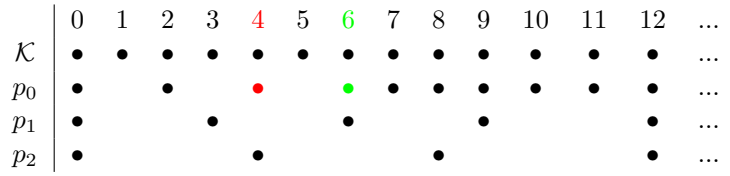


Figure 4.6: A platform dynamic behavior.

$\{60MHz, 120MHz\}, \{40MHz\}$ and $\{30MHz\}$ respectively. Suppose the frequency switch of p_0 is one cycle. The dynamism of the platform is managed by a controller which imposes the scenario $(60MHz, 40MHz, 30MHz)$ at the beginning for p_0, p_1 and p_2 respectively. Then, it adapts the scenario to a new one ($120MHz$, $40MHz$, $30MHz$) at instant 4. The second behavior only takes effect at instant 6 as the switch cost of p_0 is assumed to be one cycle here.

4.3.3 Application Mapping on Platforms

Mapping is one of the key design steps in embedded system co-design process.

Definition 4 (mapping)**.** *Given an application composed of a task set T and an execution platform composed of a processing element set P, a mapping of the application onto the platform is defined as a total function $M : T \to P$.*

In order to evaluate system designs, elementary costs including time and energy costs of all events, should be provided. Such values are usually fixed and obtained statically by either profiling each task execution on the potential processing elements, or analyzing the worst-case execution times (WCET) and energy consumptions. However, different profiling might get different results due to uncertainty, and worst-case values can result in very pessimistic performance analysis values.

Our framework rather considers the time cost of each event e as *an interval* denoted by $[\alpha_\perp, \alpha_\top]$, with lower and upper bounds represent typically the *best and worst case execution times* respectively. Selection strategies such as defining a probability distribution on the values of this interval can be defined so as to capture realistic values at simulation time. The event energy cost is defined similarly. The event time cost values are specified in terms of number of clock cycles of mapped PEs, and energy costs are specified as the profiling values on PEs. Notice that the same event in different behaviors may have different cost values.

4.4 Scheduling and Design Analysis

We firstly introduce the clock modeling of application executions on platforms in our framework. The notion of *admissible scheduling* for applications in our framework is then introduced, before we propose an iteration-based scheduling algorithm for admissible scheduling. Afterwards, performance analysis based on abstract clocks to assess design choices is discussed.

4.4.1 Clock Modeling of System Executions

We introduce a ternary abstract clock encoding of application executions on platforms. Task execution on PEs and data transmission on the interconnect are considered separately. We

assume that the input data of a task are always stored in the local memory of its mapped PE, and the time cost of reading data from a local memory is neglected for the sake of simplicity.

A *ternary abstract clock* is a three-valued string over $\{-1, 0, 1\}$. The values 1 and 0 respectively represent the *active* and *idle* instants of task executions on PEs or data transmissions on interconnects w.r.t. the reference clock $\mathcal{K}$. The meaning of the value -1 depends on the context: a sequence of -1 means active at these instants if it is preceded by 1, otherwise it denotes idle.

Figure 4.7 shows an execution example encoded by ternary clocks. It captures the executions and communications of the application of Figure 4.5 on two PEs, p_0 and p_1, linked by an interconnect $\mathcal{I}$, of the platform of Figure 4.6. Here, we consider an application behavior with two iterations b_1, b_2, and a static platform behavior with frequencies of p_0 and p_1 are $60MHz$ and $40MHz$. Suppose each application event takes one PE clock cycle for execution, and the data transmissions of events e_0, e_1 respectively take one and two ticks of $\mathcal{K}$. The clock $clk(t/p)$ is used to represent the execution of task t on PE p, while $clk(t/\mathcal{I})$ represents the transmission of data produced by t on the interconnect $\mathcal{I}$. The value 1 indicates the logical instant at which an execution or a data transmission action related to an event starts on its mapped PE or interconnect portion. The sequence of -1's following 1 denotes the action duration. Similarly, value 0 indicates the instant at which an event is waiting for actions.

	0	1	2	3	4	5	6	7	8	9	10	11	12
p_0	•		•		•		•		•		•		•
$clk(t_0/p_0)$	**1**	-1	**1**	-1									
$clk(t_0/\mathcal{I})$	0	-1	**1**	0	**1**	-1							
p_1	•			•			•			•			•
$clk(t_1/p_1)$	0	-1	-1	**1**	-1	-1	**1**	-1	-1				
$clk(t_2/p_1)$	0	-1	-1	-1	-1	-1	-1	-1	-1	**1**	-1	-1	

Figure 4.7: Execution encoding with ternary clocks.

Let us consider task t_0 consisting of two events e_0 and e_1. The clocks $clk(t_0/p_0)$ and $clk(t_0/\mathcal{I})$ show that e_0 starts at instant 0, takes one cycle and finishes at instant 2, at which instant its produced data starts to be transmitted, and the transmission takes one tick. The event e_1 starts at instant 2 and finishes at 4, at which instant its produced data is transmitted, and arrives at instant 6 after 2 ticks. The executions of events e_2 and e_3 of task t_1 start after the instants when the data transmissions of their preceding events are completed. Task t_2 can only start at instant 9, as *it* shares p_1 with task t_1 and p_1 is running t_1 at instant 6 when its input data are ready. $clk(t_1/\mathcal{I})$ and $clk(t_2/\mathcal{I})$ are empty and omitted here as tasks t_1 and t_2 do not produce any data. Indeed, as mentioned in the beginning of the current section, all tasks are assumed to read their data from a local memory, which does not involve the interconnect. Only a flavor of ternary clocks is given here, more aspects including the red "1" will be explained later.

For an efficient symbolic manipulation, ternary clocks can also be represented in a compact way. For instance, in Figure 4.7, the ternary clock $clk(t_1/p_1)$ is written as $0(-1)^2 1(-1)^2 1(-1)^2$, where the exponent denotes the number of repetitions or periods.

4.4.2 Admissible Scheduling of Applications

The following three requirements are considered for characterizing an admissible scheduling:

i) *precedence relation preservation*: the precedence constraints between events defined in application behaviors are preserved,

ii) *non-simultaneous execution*: the executions of two or more events at the same time are not allowed on the same processor,

iii) *cycle integrity*: when the activation instants of task events and the clock instants or ticks of their executing processors do not coincide, the non null delay between these instants is fully taken into account, i.e., processors are not yet ready to execute the events. For such an event, its effective execution is postponed to the next processor clock tick from its current position in time.

Requirement *i*) should be further refined as follows when taking into account communication aspect: 1) the data transmission of an event cannot start before the event ends, and 2) if event e_1 is preceded by another event e_2 (e_1 and e_2 do not belong to the same task), then e_2 cannot start before event e_1's data transmission ends.

In literature, such as in [Lee & Messerschmitt 1987], only the first two requirements are considered for admissible scheduling. Requirement *iii*), i.e., the possible non null delays between activation instants of task events and clock ticks is not taken into account. An example of execution scenario illustrating the issue raised by point *iii*) is shown in the scheduling of t_1 on p_1, denoted by the italic $clk(t_1/p_1)$ in Figure 4.8. Compared to Figure 4.7, here we suppose the transmission for the produced data of e_0 takes 2 ticks. In this case, the produced data of e_0 starts its transmission at instant 2, and finishes the transmission at instant 4 of the reference clock. The task t_1 then starts its execution at this instant, denoted by the first red colored "1", which lies between two clock ticks of processor p_1. This is not a valid execution. The execution of task t_1 should be postponed to some instant which coincides with a clock tick of p_1. The bold $clk(t_1/p_1)$ gives a valid scheduling where the first red "1" is postponed to instant 6. Similarly, the second red "1" also violates the third requirement, and is postponed in the bold $clk(t_1/p_1)$ to instant 9.

	0	1	2	3	4	5	6	7	8	9	10	11	12
p_0	•		•		•		•		•		•		•
$clk(t_0/p_0)$	**1**	-1	**1**	-1									
$clk(t_0/\mathcal{I})$	0	-1	**1**	-1	**1**	-1							
p_1	•			•			•			•			•
$clk(t_1/p_1)$	0	-1	-1	-1	1	-1	-1	1	-1	-1			
$\mathbf{clk(t_1/p_1)}$	0	-1	-1	-1	-1	-1	**1**	-1	-1	**1**	-1	-1	

Figure 4.8: Execution encodind with ternary clocks.

4.4.3 Scheduling Algorithm

We propose an iteration based admissible scheduling algorithm for application executions on DFS-enabled MPSoCs. An "iteration" refers to the iterative application executions, and the scheduling algorithm schedules tasks iteratively. It supposes that tasks are statically allocated and executed on mapped PEs, and the execution of events are non-preemptive. Given an application behavior b_T, the events that have no precedence relations need to be ordered by defining a scheduling order denoted by $\phi(b_T)$. This order is used to indicate the priority when events are competing for the same PE. Algorithm 1 gives the scheduling algorithm.

The inputs of the algorithm are as follows:

Algorithm 1 Scheduler

for all $t \in T, p \in P$ **do**
  $clk(t/M(t)) = \oslash, clk(t/\mathcal{I}) = \oslash$;
  $clk(p) = \oslash$;
end for
$pre\_b_P = \mathcal{C}(B_P)$;
for $i = 1 \rightarrow \omega$ **do**
  if $\mathcal{C}(B_P) \neq pre\_b_P$ **then**
    add costs to those PEs that have changed frequencies;
    $pre\_b_P = \mathcal{C}(B_P)$;
  end if
  for all $e_j \in \varphi(\mathcal{C}(B_T))$ **do**
    if $pre(e_j) \bigcap \varphi = \oslash$ **then**
      $rp = 0$;
    else
      $rp \leftarrow \max\{|clk(e.t/\mathcal{I})||\forall e \in pre(e_j) \bigcap \varphi\}$;
    end if
    $execute(e_j, M(e_j.t), rp)$;
  end for
end for
function *execute(Event e, PE p, int rp)*
if $rp > |clk(p)|$ **then**
  $sp \leftarrow \min\{index|index/n_r(p) = 0, index > rp\}$;
  $clk(p) = clk(p) \oplus 0(-1)^{[sp-|clk(p)|-2]}$
end if
if $|clk(e.t)| < |clk(p)|$ **then**
  $clk(e.t) = clk(e.t) \oplus 0(-1)^{[|clk(p))|-|clk(e.t)|-1]}$
end if
$clk(e.t) = clk(e.t) \oplus 1(-1)^{[(execSel(e,\mathcal{C}(B_T)))*n_r(p)-1]}$;
$clk(p) = clk(p) \oplus 1(-1)^{[(execSel(e,\mathcal{C}(B_T)))*n_r(p)-1]}$;
if $e.data = 0$ **then**
  return;
end if
if $|clk(e.t/\mathcal{I})| < |clk(e.t/p)|$ **then**
  $clk(e.t/\mathcal{I}) = clk(e.t/\mathcal{I}) \oplus 0(-1)^{[|clk(e.t/p))|-|clk(e.t/\mathcal{I})|-1]}$;
end if
$minBW \leftarrow \min\{bandwidthSel(p, p')|\forall p' \in P, \exists e' \in suc(e) \bigcap M(e') = p'\}$
$clk(e.t/\mathcal{I}) = clk(e.t/\mathcal{I}) \oplus 1(-1)^{[(e.data/minBW)*f_{\mathcal{K}}-1]}$;
end function

- a dynamic application behavior $(\mathcal{C}(B_T), \omega)$,
- dynamic platform behavior $\mathcal{C}(B_P)$, and best and worst transmission bandwidths $\beta_\perp$ and $\beta_\top$ for all PE pairs;
- a mapping function M;
- scheduling orders $\phi(b_T), \forall b_T \in B_T$;
- best and worst case execution times $\alpha_\perp$ and $\alpha_\top$ on possible mapped PEs for all events associated with different behaviors;
- online selection strategies $execSel(e, b_T)$ and $bandwidthSel(p_i, p_j)$ to select an execution time value of e in behavior b_T and a bandwidth value for transmitting data from p_i to p_j according to the best and worst values respectively.

The algorithm outputs are the scheduling clocks $clk(t/M(t))$, $clk(t/\mathcal{I})$, $\forall t \in T$ and $clk(p), \forall p \in P$. PE scheduling clock represents the execution behavior of all tasks mapped on this processor. If there is only one task running on it, It is the same to the task's scheduling clock. When two or more tasks are mapped on the same PE, it is equal to the composition of the clocks of all tasks running on it. E.g., the scheduling clock of p_1 in Figure 4.7 would be the composition of $clk(t_1/p_1)$ and $clk(t_2/p_1)$, that is $0(-1)^2 1(-1)^2 1(-1)^2 1(-1)^2$.

In the algorithm, $e.t$ is used to represent the source task of e, and $\oplus$ is a binary concatenation operator operates on clocks, which appends the right operand to the end of the left one. For example, $1(-1)^5 \oplus 1(-1)0(-1) = 1(-1)^5 1(-1)0(-1)$.

From line 1 to 5, the algorithm initializes the scheduling clocks to empty, and the platform behavior memory pre_b_P to the current platform behavior according to $C(B_P)$. Lines 6 to 19 specify the iteration scheduling loop. At iteration i, the algorithm firstly checks whether the platform behavior has changed compared to the previous iteration. Frequency adaptation costs are added to the scheduling clocks of adapted PEs (as done in Figure 4.6), and the platform behavior memory is evaluated to the current one if it has.

Next, the scheduler schedules all the events of the current application behavior $C(B_T)$ on their mapped PEs following the queue φ (lines 11 to 18). Given a behavior b_T, $\varphi(b_T)$ is constructed by ordering all its events according to $\phi(b_T)$ and the event precedence relations. $pre(e_j)$ represents the set of events preceding e_j. To ensure admissible scheduling, it takes care of the three admissibility requirements by using variables rp and sp to preserve precedence relation and cycle integrity respectively, and the incremental construction of PE scheduling clocks to avoid the simultaneous executions of events. rp computes the instant w.r.t. the reference clock where all the data produced by the events preceding the event have just arrived (line 15), and is evaluated to 0 if e_j has no precedent events (line 13). $|clk|$ represents the length (i.e., number of instants) of clk. sp is used when the PE is idle at rp, and computes the very next instant respecting cycle integrity of the corresponding PE after rp (line 22). $n_r(p)$ represents the number of reference clock ticks that the cycle of PE p occupies in the current behavior. sp represents the earliest instant at which the PE can run an event. Between rp and sp, the event has to wait, denoted by 0 followed by -1's (line 23).

The scheduling clock of the task must coincide with the schedule of its mapped PE, i.e., the schedule of task event e should be the same on both scheduling clocks. This is ensured by the statements from line 25 to line 27: the algorithm looks at the instant where the PE is about to run the event, and appends 0 followed by a sequence of -1s to the task scheduling clock if it needs to wait (e.g., some other event is executing on the PE). The

scheduling clocks of the task and PE is then updated by appending the scheduling clock of the event at lines 28 and 29 respectively. Lines 30 to 37 deal with the updating of the scheduling clock $clk(e.t/\mathcal{I})$. Nothing is done if the event produces no data (line 30). $e.data$ represents the size of e's produced data. Otherwise, the scheduler firstly ensures that the data transmission of the event cannot start before the instant at which the event ends (lines 33 to 35), and then appends the schedule of the data transmission to the clock (line 37). $minBW$ represents the minimal bandwidth between the mapped PE p of the event e and the PEs that execute at least one event that requires its produced data as input (line 36). $suc(e)$ represents the set of such succeeding events of e. Using the minimal value ensures that when event e has more than one successors, i.e., sending data to more than one events, its data transmission finishes until its data have arrived to all successors.

Property 1 (Correctness)**.** *Algorithm 1 always generates an admissible schedule in the sense of the three requirements given in Section 4.4.2.*

Proof. To prove this property, we need to prove that the following three requirements are met: the precedence relation preservation, cycle integrity for all scheduling clocks of tasks on PEs i.e., $clk(t/M(t)), \forall t \in T$, and scheduling clocks of PEs i.e., $clk(p), \forall p \in P$, and non-simultaneous execution schedules exist for the same processor.

Precedence relation preservation. We distinguish the precedence relations defined a) on events within the same task behavior, and b) on events from different task behaviors. For case a), $\forall e_j^p, e_j^q, e_j^p \prec e_j^q$, the precedence relation is preserved due to the fact that the algorithm schedules e_j^p before e_j^q and the scheduling clocks are computed incrementally, i.e. the schedule of e_j^q is appended to the corresponding scheduling clocks after the one of e_j^p. For case b), $\forall e_i^p, e_j^q, i \neq j, e_i^p \prec e_j^q$ is kept due to, on the one hand, the algorithm projects e_j^p before e_j^q because of the order of ϕ, and on the other hand, the schedule of e_j^q starts after the finish instant the schedule of e_j^p which is under the charge of the variable rp.

Cycle integrity. This requires that the generated scheduling clocks $clk(t/M(t)), \forall t \in T$ and $clk(p), \forall p \in P$ have 1s and 0s only coincide with corresponding processor tick instants. Since these clocks are only updated in Lines 23, 26, 28, 29 of Algorithm 1, we need to prove these updates do not introduce violation. Initially, all clocks are empty, thus if the codes of Lines 23 and 26 are executed, the 0 would be appended to the first instant which apparently coincides the clock tick. Moreover, after a number of -1s are appended, each clock has the next instant coinciding with the clock tick due to sp for Line 23 and $sclk(p)$ for Line 26 (whose next instant is a clock tick). Then for Lines 28 and 29, each 1 is appended to a clock instant coinciding with clock tick, and each clock finishes just before a tick instant as the schedule of an event spans a number of processor cycles. As a result, the property follows for all these scheduling clocks.

Non-simultaneous execution schedules. This property follows due to the following two facts of our algorithm. Firstly, before scheduling an event e, the algorithm always matches its corresponding task and processor scheduling clocks, which ensures that the schedule $clk(e)_{pos}$ of e is the same on both clocks w.r.t. the reference clock. Secondly, the processor scheduling clock is computed incrementally, which ensures that at the same instant at most one event could be scheduled. □

4.4.4 Performance Analysis

Based on the previous scheduling algorithm, the computation of a number of performance parameters are possible. We have integrated the following performance parameter computation in our framework.

- execution time of a task t_i: $ET(t_i) = |clk(t_i/M(t_i))| * (1/f_{\mathcal{K}})$, and PE p_i: $ET(p_i) = |clk(p_i)| * (1/f_{\mathcal{K}})$;

- usage ratio of a PE p_i: $UR(p_i) = nbc(clk(p_i)) \div (nbc(clk(p_i))+nic(clk(p_i)))$, where $nbc(clk(p_i))$ and $nic(clk(p_i))$ represent respectively the number of busy and idle cycles of clock $clk(p_i)$.

- energy consumption of a PE p_i: $EC(p_i) = \sum EC(t_j) + nic(clk(p_i)) * iec(p_i)$, where $M(t_j) = p_i$, $EC(t_j)$ represents the accumulated energy costs of task t_j during simulation, and $iec(p_i)$ represents the energy cost of an idle cycle of p_i;

- used local memory space for communicating tasks t_i, t_j connected by channel $ch(t_i, t_j)$: $MS(ch(t_i, t_j))$. It is assumed that an event does not produce (resp. remove) its output (resp. input) data to (resp. from) the memory until its execution is finished. $MS(ch(t_i, t_j))$ is computed by keeping track of the data production and consumption operations on $ch(t_i, t_j)$ during simulation. Two variables are used: the first one for keeping track of the current stored data size, and the second one for recording the maximal stored data size during simulation. $MS(ch(t_i, t_j))$ is equal to the value of the second variable after simulation finishes. This storage space value gives an idea of how much buffer space can be reserved for their communication when implementing the mapping and scheduling.

As a general remark, the proposed scheduling algorithm based on the clock modeling framework is flexible enough to help designers to efficiently assess several design aspects at an early design stage. The idea is to play with the values of concerned parameters of the algorithm and fix the rest.

4.5 Design Space Exploration

We build on top of the scheduling algorithm an exploration loop to target optimal software/hardware mapping and platform configuration (i.e., frequency values of PEs) solutions. Two exploration methods are employed: an exhaustive one and an heuristic one. The objective is to find out a set of design points optimizing the time and energy consumptions, or the so-called Pareto optimal solutions [Zitzler 1999].

The exploration algorithms takes the same inputs as Algorithm 1, except the mapping parameter which is to be explored. Moreover, the platform manager should be set to always use the same explored platform behavior during the application scheduling. The output is a set of Pareto optimal solutions, and each solution is a task to PE mapping and a frequency value for each PE. In the next, we describe these two exploration methods.

Algorithm 2 gives the exhaustive exploration algorithm using two recursive functions. It starts the exploration at line 1 by calling recursive function *exploreFreqCombination* with argument 1. This function is used to explore all the PE frequency combinations, i.e., all the platform behaviors. The termination condition is met when the function is called for the $(|P|+1)^{th}$ time (lines 3 to 5). Lines 6 to 12 enumerate all the possible frequency values one by one for the i^{th} PE. After each evaluation at line 7, frequency values are enumerated for the $(i+1)^{th}$ PE (line 8). We use $f(p_i)$ to denote the current frequency value of p_i. After the call is returned, if this call is for the enumeration of the last PE (i.e., line 9 is met) the recursive function *exploreMapping* is invoked with argument 1 to ask the function to recursively enumerate all the possible mappings of tasks on the current frequency values of PEs (i.e., platform behavior) from the first task. The structure of the second function is

the same as the first, and not explained here. Lines 22 to 24 are the steps to follow once a mapping choice is computed regarding a platform behavior.

Algorithm 2 Exhaustive exploration

```
1: exploreFreqCombination(1);
2: function exploreFreqCombination(i)
3: if i = |P| + 1 then
4:    return;
5: end if
6: for all f_j ∈ fs(p_i) do
7:    f(p_i) ← f_j;
8:    exploreFreqCombination(i + 1);
9:    if i = |P| then
10:      exploreMapping(1);
11:   end if
12: end for
13: end function
14: function exploreMapping(k)
15: if k = |T| + 1 then
16:   return;
17: end if
18: for all p_i → |P| do
19:   M(t_k) ← p_i
20:   exploreMapping(k + 1);
21:   if k = |T| then
22:      1) invoke Algorithm 1 with the current mapping and platform behavior;
23:      2) compute the system time and energy costs as in Section 4.4.4;
24:      3) check whether the current mapping and platform behavior is a Pareto optimal
         solution: save it if it is, otherwise continue;
25:   end if
26: end for
27: end function
```

As the design space of the problem grows exponentially, the use of the exhaustive exploration is limited. A common solution is thus to trade optimality for speed, and uses heuristic techniques. We consider evolutionary algorithms (EAs) [Zitzler 1999], as such algorithms have the ability to find multiple Pareto-optimal solutions in one single run. EAs use mechanisms inspired by biological evolution, such as selection and variation. They operate on a set of candidate solutions (called population), and evolve them iteratively. Solutions are assessed by a cost function (called fitness function), and better solutions are more likely to survive. In our framework, we consider the well-known evolutionary algorithm NSGA-II [Deb *et al.* 2000]. To combine it with our framework, we have considered its implementation in the JMetal framework [Durillo & Nebro 2011]. The following adaptations have been done based on our framework to apply the algorithm:

- represent the exploration space by $|T|$ double variables, each variable corresponding to the mapping of one task with the integer part representing the index of its mapped PE, and fraction part for the PE frequency index;
- define two objectives: minimizing the time and energy consumptions;

- evaluate for computed solutions the values of the two objectives: the values are obtained by 1) invoking the scheduling algorithm (Algorithm 1) with the corresponding solutions; 2) computing the time and energy costs.

4.6 Implementation and Experiments

We give an overview of the implementation of our framework, then we show some experimental results.

4.6.1 CLASSY Tool

The modeling, scheduling and analysis framework presented in this paper have been implemented in a prototype tool, named CLASSY (CLock AnalySis SYstem). The two design space exploration methods: exhaustive and heuristic-based exploration have also been integrated in the tool. An automatic transformation from MARTE described system specifications to our reasoning framework, however, is still under construction. Some inspirations will be taken from [Abdallah 2011].

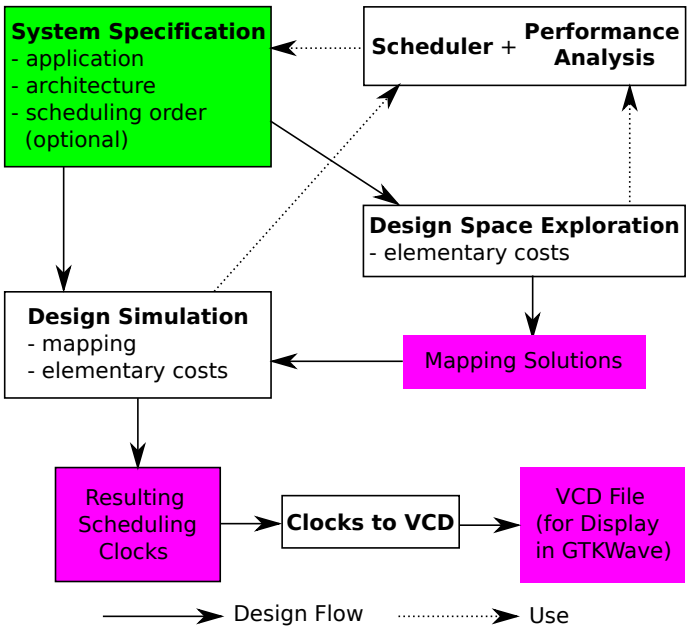


Figure 4.9: Overview of the CLASSY tool.

Figure 4.9 shows the overview of the Classy tool. It has around two thousand Java code lines and consists of five modules as follows:

- *system specification* provides interface for the user to define (dynamic) application behavior including task behaviors, precedence relation between events, the associated controller, (dynamic) execution platform behavior including PEs, their frequency values, transmission bandwidths between PEs, the associated controller, and the scheduling order for those events that have no precedent relations between each other (this is optional, if not defined, the tool follows the order that the events are defined).

- *scheduler and performance analysis* performs scheduling as given in Algorithm 1, and performance analysis as in Section 4.4.4 w.r.t. the system specification, a mapping choice and corresponding elementary costs. It results in an admissible schedule which is a set of scheduling clocks including the scheduling clocks of all tasks as well as the composed scheduling clocks of processors, and the execution time, energy consumption, etc.

- *design simulation* provides an interface to allow the designer simulate and analyze his/her mapping choice. The elementary costs w.r.t. the mapping should also be provided. It uses the *scheduler and performance analysis* module to perform simulation and analysis.

- *clocks to vcd* translates the scheduling clocks of tasks and PEs into a *vcd* file, so that it can be feed to the GTKWave tool to to visualize the schedules of tasks on their mapped processors, as well as the running states of processors.

- *design space exploration* implements the two exploration methods in Section 4.5 to perform design space exploration and generate Pareto-optimal mapping solutions. The elementary costs w.r.t. all the possible mappings should be defined. It uses the *scheduler and performance analysis* module to perform simulation and analysis of each mapping choice. During the exploration, the scheduling clocks are not computed in order to make the exploration more efficient. It results in a set of Pareto-optimal mapping solutions characterized by their performance results. If the user wants to visualize some interesting mapping solution, he/she can then follow the path from the *design simulation* module to the *clocks to vcd* module to generate the vcd file.

4.6.2 Experimental results

We have performed a number of experiments to show the usability and flexibility of our framework. We also evaluate its efficiency and scalability.

In the first experiment *Exp.*1, we consider an implementation of the M-JPEG decoding algorithm described in Section 4.2 in the SoCLib environment [SoClib 2012], dedicated to SoC prototyping and cycle-accurate simulation. The results are compared with those observed with CLASSY to see the precision.

In the second experiment *Exp.*2, we consider reference application models from those used in [Stuijk *et al.* 2008] and compare our results with them. Their work performs a throughput and buffering trade-off exploration based on cyclo-static and synchronous dataflow graphs (CSDFGs and SDFs). Reasoning at this abstraction level allows designers to quickly predict the timing behavior of an application and required buffer size before realizing it. The buffer size is the main concern of the throughput analysis. Our framework also advocates high level reasoning and analysis, but has different concerns and brings different insights for designers. In comparison with the results of [Stuijk *et al.* 2008], we discuss and explain the differences.

In the third experiment *Exp.*3, we employ the MP3 decoder from [Stuijk 2007] to evaluate the scalability of our framework.

In the fourth experiment *Exp.*4, we use a case study from [Stuijk 2007] to show how our framework can be used to assist a designer in embedded system design, even though the case study has different design objectives. We also compare the results to see the precision.

At last (*Exp.*5), we use the example M-JPEG decoder and a NoC based MPSoC to exhibit the flexibility of our framework, particularly the ability to capture system dynamic

behavior, and allow customized communication and computation simulations. All experiments are performed on a laptop with a Intel Core 2 Duo CPU of 2.4GHz and a 2Go main memory.

*Exp.*1 : **Analysis precision w.r.t. low level cycle-accurate simulation environment**

We consider the implementation [Boumedien 2011] of the M-JPEG decoding algorithm in the SoCLib environment [SoClib 2012], dedicated to SoC prototyping and cycle-accurate simulation. The obtained results are compared with those observed with CLASSY. The simulation results observed with our tool are correct-by-construction and obtained rapidly at a low cost.

Configurations identifiers	M-JPEG tasks	Mapped processors
1	Demux, Vld, Iqzz, Idct, Libu	p_1
2	Demux, Vld, Iqzz	p_1
	Idct, Libu	p_2
3	Demux, Iqzz, Libu	p_1
	Vld, Idct	p_2
4	Demux, Vld	p_1
	Iqzz	p_2
	Idct, Libu	p_3
5	Demux, Iqzz	p_1
	Vld, Libu	p_2
	Idct	p_3
6	Demux	p_1
	Vld	p_2
	Iqzz	p_3
	Idct, Libu	p_4
7	Demux, Libu	p_1
	Vld	p_2
	Iqzz	p_3
	Idct	p_4
8	Demux	p_1
	Vld	p_2
	Iqzz	p_3
	Idct	p_4
	Libu	p_5

Table 4.1: Analyzed mapping configurations for M-JPEG.

Experimental setup. For the execution of the M-JPEG decoder, a multiprocessor platform with a shared multi-bank memory, which can be configured to support up to five processors interconnected by using a bus or a network-on-chip (NoC), is considered in [Boumedien 2011].

Concerning the application/platform mapping, a buffer is mapped on the memory bank associated with its consumer task. When two communicating tasks are mapped on different processors, a task consumes its data from its memory bank and writes produced data on the memory bank of the processor executing the consumer task. The studied mapping configurations are summarized in Table 4.1. Up to five processors $\{p_1, p_2, p_3, p_4, p_5\}$ are

considered. For instance, in configuration number 1, all M-JPEG tasks are executed on processor p_1 while in configuration number 2, the successive tasks Demux, Vld, Iqzz are executed on p_1 and the successive tasks Idct and Libu are executed on p_2. In configuration number 3 also the same processors are considered, but the defined mapping does not select successive tasks to execute on the same processor. We refer to multiprocessor configurations like the number 2 as *successive task mappings* and multiprocessor configurations like the number 3 as *non successive task mappings*.

Abstract clock-based design. The behavior of the M-JPEG application is composed of two static behaviors: 1) an initialization part, indicated by a blue curve, where some initial communications are achieved between the Demux task and the Vld and Iqzz tasks; and 2) a periodic part, indicated by a red curve, which is repeated 36 times and consists of pixel block-wise decoding of an image.

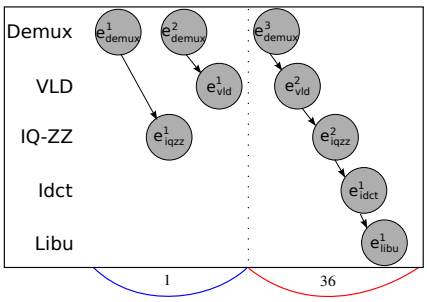


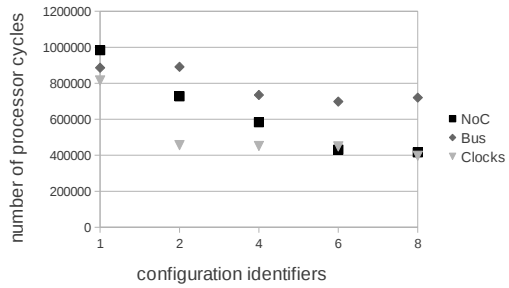
Figure 4.10: Application behavior for M-JPEG.

Regarding the input profiling data for each task of the M-JPEG, Table 4.2, taken from [Boumedien 2011] gives the time cost values in terms of processor cycles corresponding to each event shown in Figure 4.10. The given values are average values obtained from the profiling of the application implementation in SoCLib. The performance properties about the communication aspect, e.g., the bandwidth, the bandwidth selection strategy are not given in [Boumedien 2011]. Here, we suppose the communication speed is fast enough, and set the communication costs to 0 in our CLASSY experiment.

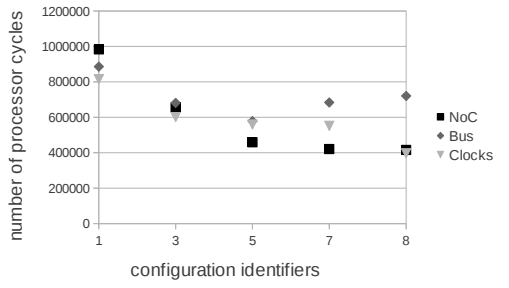
Tasks	Observed events	Number of repetitions	Number of processor cycles
Demux	e^1_{demux}	1	12651
	e^2_{demux}	1	21032
	e^3_{demux}	36	2464
Vld	e^1_{vld}	1	28042
	e^2_{vld}	36	3007
Iqzz	e^1_{iqzz}	1	1668
	e^2_{iqzz}	36	4946
Idct	e^1_{idct}	36	8978
Libu	e^1_{libu}	36	1496

Table 4.2: Profiling data about M-JPEG tasks as inputs for CLASSY.

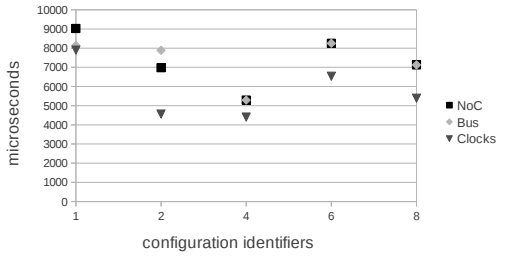
Comparison of simulation results. A part of the simulation results obtained from our clock-based approach on the M-JPEG are reported in Figure 4.11, together with those observed with SoCLib. They represent the temporal performances associated with the mapping configurations summarized in Table 4.1. In Figures 4.11(a) and 4.11(b) all pro-



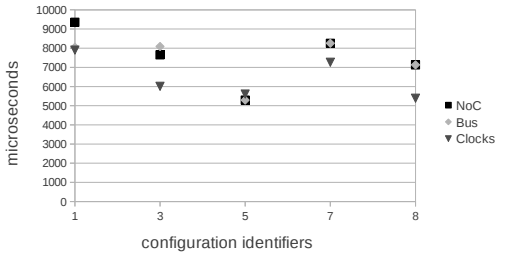
(a) Successive task mappings (processors with the same frequency)



(b) Non successive task mappings (processors with the same frequency)



(c) Successive task mappings (processors without the same frequency)



(d) Non successive task mappings (processors without the same frequency)

Figure 4.11: Execution times for M-JPEG decoder on an image: CLASSY *vs* SoCLib cycle-accurate simulations (communication via bus and NoC).

cessors always operate at the same frequency, while it is not the case in Figures 4.11(c) and 4.11(d). Two system implementations are considered in SoCLib according to the communication infrastructure: bus *versus* NoC.

The experiments show that our clock-based approach yields results with similar tendency as those obtained with SoCLib. The precision of the results provided by CLASSY appears good when compared to the NoC-based results. However, it is not the case when considering the bus-based results. This observation is explained by the fact that NoCs offer higher communication performances than buses. The execution time obtained with NoCs is therefore shorter thanks to reduced communication time. In addition, possible bus access conflicts, which increase the communication overhead, lead to lower performances compared to NoC-based implementations. This issue is usually observed when the number of processors sharing the same bus gets higher. This may explain the increase of the execution time in Figure 4.11(b), from configuration number 5 (three processors) to configuration number 8 (five processors). Since the input profiling data given in Table 4.2 do not cover such communication overheads and we set the communication costs to 0 in our CLASSY experiment, the obtained results are less precise w.r.t. bus-based implementations.

*Exp.*2 : **High level design analysis**

We use five application models from [Stuijk *et al.* 2008] as benchmark set for this experiment: H.263 decoder, MP3 decoder, a bipartite SDFG, Example SDFG and Example

CSDFG. In their work, they aim to analyze the exact minimum buffer bounds for any achievable throughput of modeled applications. *Storage distribution* is defined in their work to represent the number of data tokens stored in each channel. The sum of the numbers is called *distribution size* of the storage distribution. Each actor is characterized by an execution time for SDFG or a sequence of execution times for CSDFG. The throughput is defined as the number of iterations per time unit. The storage distribution is the only factor that influences the task executions. Its analysis abstracts away the communication time costs and execution platform, and allows the concurrent firings of the same task. It favors a rapid design exploration, however, needs to be combined with a particular design flow for implementation [Stuijk 2007].

Mapping is one key design decision in system implementation. Our framework favors the efficient exploration of mappings, while throughput and storage distribution are also analyzed. We tune the inputs of our framework as follows, and then compare and discuss our experimental results w.r.t. [Stuijk *et al.* 2008]:

- The communication cost is set to 0 so as to ignore communication costs;
- The number of PEs is set to the number of application tasks; all PEs are set to have the same frequency value; task execution times are set to the values as in their work, and they are the same on all PEs;
- The data token production and consumption operations are defined as in the graphs;
- Task energy cost that is irrelevant in the experiment is set to a random value, and ten iterations are simulated for each application.

Tables 4.3 and 4.4 show respectively the results of the reference work and our experimental results. The exhaustive exploration method has been employed for the analysis of the first four examples, and the heuristic algorithm is employed for the MP3 decoder by setting population size to 10 and number of iterations to 10^6.

We distinguish three reasons that lead to the throughput difference for all the examples. First, our framework does not allow any concurrent firings of the same task (i.e., the firing of a task can only be on its mapped PE), while the reference work does unless disallowed in the application graph. Second, our framework assumes that buffer storage spaces are enough for communications, while the reference work has a storage space bound during scheduling. Third, our framework performs simulations for finite number of iterations and compute the throughput based on the complete simulation, while the reference work assumes infinite number of iterations, and computes the throughput based on the periodic part of a constructed labelled transition system state space. Our framework does not investigate the possible periodic execution behavior, as it targets fast simulations and simulation time cost evaluations. The results for Example SDFG, Example CSDFG and H.263 decoder are close, as they have some tasks that does not allow concurrent firing. Results for Bipartite and MP3 decoder have quite different results, as most of the actors have high concurrent firing possibilities. Globally, as we can see from Tables 4.3 and 4.4, even thought assuming enough storage space in our framework, the maximal achievable throughputs for all applications are still smaller, as well as the minimal throughputs. This implies that the concurrency level plays a more important role than buffer size w.r.t. maximizing throughput in these applications.

	Example SDFG	Example CSDFG	Bipartite	H.263 Decoder	MP3 Decoder
#tasks/channels	3/2	3/2	4/4	4/3	14/14
max. throughput	0.25	> 0.08	0.06	1×10^{-4}	6.88×10^{-6}
distribution size	10	16	35	8006	54
min. throughput	0.143	< 0.06	0.04	5×10^{-5}	5.33×10^{-7}
distribution size	6	11	28	4753	22
#Pareto points	9	5	8	3254	17
Run-time	1ms	-	1ms	53mins	10.7s

Table 4.3: The experimental results taken from [Stuijk *et al.* 2008]. The unit of throughput is the number of iterations per time unit. The unit of distribution size is the number of data tokens. The results of the example CSDFG is given in a graph, where the maximum and minimum throughputs are not easier to recognized precisely. $>$ and $<$ means respectively sightly bigger and smaller than. The run-time of the experiment is not given.

	Example SDFG	Example CSDFG	Bipartite	H.263 Decoder	MP3 Decoder
#tasks/channels	3/2	3/2	4/4	4/3	14/14
max. throughput	0.227	0.08	4.0×10^{-3}	8.8×10^{-5}	2.66×10^{-7}
distribution size	21	24	164	7410	160
# used PEs	3	3	4	4	12
usage ratio(%)	(68.2,90.1,45.5)	(96,64,40)	(14.2,99.5,46.2,31.6)	(88.1,20.9,83.7,35.2)	λ
min. throughput	0.111	0.04	2.1×10^{-3}	3.9×10^{-5}	1.23×10^{-7}
distribution size	8	24	86	4753	126
# used PEs	1 PE	1 PE	1 PE	1 PE	1PE
usage ratio(%)	(100)	(100)	(100)	(100)	(100)
#Pareto points	4	5	7	6	10
Run-time	84ms	191ms	728ms	4.8s	14mins58s

Table 4.4: The experimental results of our framework. The unit of throughput is the number of iterations per time unit. The unit of distribution size is the number of data tokens. λ = (3.8,99.1,99.1,0.02,3.9,0.26,0.26,2.8,4.4,0.39,1.84,1.84).

Regarding the run-time difference, two reasons are distinguished. First, different computers have been used to perform experiments. Second, our framework explores the mapping space, while [Stuijk *et al.* 2008] explores the storage distribution space. In our case, if the application has N actors/tasks and the platform is composed of M PEs, and each actor can be mapped on any PE, the mapping space is M^N. This can be observed from Tables 4.3 and 4.4. Both tools scale well for the first three applications, as they have small number of tasks and storage distribution sizes. The run-time in Table 4.3 gets larger for the H.263 decoder (53 minutes), as the application requires a distribution size of thousands of tokens. As it only has four actors, our framework scales well. However, when considering the MP3 decoder, the mapping space has an order of magnitude 10^{16}. Suppose the analysis of one mapping takes $1ms$, then exploring the complete space would take thousands of years! The exhaustive exploration is thus infeasible. We have taken the MP3 encoder example from [Stuijk 2007], and use the SDF^3 tool [Stuijk *et al.* 2006a] to compute the results, as we did not find the MP3 encoder SDF model in [Stuijk *et al.* 2008]. This example is also used in the next section to show the scalability of the heuristic exploration.

The computed storage distribution sizes are also given in Table 4.4. The results from [Stuijk *et al.* 2008] are known to be optimal in their context. On the contrary, our framework does not target the storage size minimization to achieve some throughput, but rather gives an idea of storage distribution that can be used for a mapping to achieve the corresponding throughput. However, this is one direction of our future work. Moreover, our framework provides information like mapping (show as used number of PEs, as PEs are identical) and the usage ratio of each PE, while this is not the case for [Stuijk *et al.* 2008].

*Exp.*3 : **Scalability**

The MP3 decoder example presented in the previous section has 14 tasks, and when mapping it onto a platform with 14 PEs, the mapping space would explode, and makes the exhaustive exploration infeasible. In general, the run-time of our tool can be estimated by multiplying the analysis time of one mapping and the size of the mapping space. The analysis time for one mapping depends on the number of task activations within one iteration and the number of simulated iterations. When the exploration time becomes very large, the heuristic exploration must be applied. In the following, we use the same MP3 decoder example, which has 14 tasks and 27 task activations within one iteration, and an execution platform composed of 14 PEs to evaluate the scalability of our heuristic algorithms.

Table 4.5 shows the results by applying the heuristic exploration w.r.t. two parameters: population size and the number of iterations. With the number of iterations grows by 10 times, the run-time also grows around 10 times. On the contrary, the population sizes does not have much influence on the run-times when iterating the same numbers of times. It shows that the heuristic exploration can significantly reduce the run-time, but does not have much influence on the results.

*Exp.*4 : **Supporting system design**

We show how our framework can support embedded system co-design by using the H.263 decoder and the NoC based MPSoC architecture from chapter 10 of [Stuijk 2007], and compare our results with [Stuijk 2007] to evaluate the precision of our framework. Design inputs including execution times of actors, PE frequencies, NoC bandwidths are taken directly from [Stuijk 2007]. The H.263 decoder is composed of four tasks: variable length decoder (VLD), inverse quantization (IQ), inverse discrete cosine transformation (IDCT) and motion compensation (MC). The architecture has four PEs to execute the decoder: two

(#populations,#iterations)	$(10, 10^3)$	$(10, 10^4)$	$(10, 10^5)$	$(10^2, 10^5)$
max. throughput ($*10^{-7}$)	2.6464	2.6526	2.6518	2.6570
min. throughput ($*10^{-7}$)	1.2853	1.2282	1.2211	1.2308
Run-time	921ms	9.58s	1min34s	1min32s
(#populations,#iterations)	$(10^3, 10^5)$	$(10, 10^6)$	$(10^3, 10^6)$	$(10, 10^7)$
max. throughput ($*10^{-7}$)	2.6578	2.6578	2.6578	2.6578
min. throughput ($*10^{-7}$)	1.2285	1.2213	1.2202	1.2202
Run-time	1min47s	14mins58s	17mins33s	2h31mins

Table 4.5: The scalability experiment. The unit of throughput is the number of iterations per time unit.

ARM7 processors, a VLD accelerator and a MC accelerator. The accelerators can execute the corresponding tasks, reducing the execution time by 50% when compared to an ARM7. All PEs operate on a 500MHz frequency, and the NoC runs at frequency 1.5GHz. The design flow reasons on time-unit, and derives that one time unit is 2 ns. It employs the *resource virtualization* technique, and the TDMA time wheels of all the PEs have thus a size of 100000 time-units. The NoC provides a predictable bandwidth: 12 bytes/time-unit. The design flow proposed in [Stuijk 2007] aims to meet a throughput constraint and meanwhile minimize the resource uses, i.e., time slices on the PE time wheels. The decoder requires a throughput 15 frames/second.

We tune our framework inputs as follows in order to compare with the results of [Stuijk 2007]. First, each PE is set to have three operating frequency values, which are selected based on the resulting time slice sizes of [Stuijk 2007]. One value is close to the converted value of the resulting time slice size, while the other two values are one around two times bigger and the other two times smaller. This allows our framework to explore the mapping on these given operating frequencies, and compute the corresponding throughputs. Second, tasks IQ and IDCT are not allowed to be mapped onto the two accelerators, and task VLD (resp. MC) is not allowed to be mapped on the MC (resp. VLD) accelerator. Third, worst-case time costs and NoC bandwidths are taken as in [Stuijk 2007], and 15 iterations are simulated. As a result, our tool computed out 13 mapping solutions with different throughput values within 10.36 seconds. The closest solution meeting the throughput constraint has the throughput 990.1 milliseconds for 15 iterations. It maps tasks VLD and MC on their accelerators with frequency values $35 * 10^4$ and $25 * 10^4$ respectively, and tasks IQ and IDCT on one ARM 7 processor with frequency 10^7. By converting the frequency values into the time slide sizes, they are 70, 50 and 2000 time units. [Stuijk 2007] has the same resulting mapping and very close time slice sizes: 71,47 and 1916 time units.

*Exp.*5 : **Dealing with dynamic behavior**

The experiments presented so far take as input worst-case computation and communication cost values. This ensures a predictable design, however may overestimate the resource requirements [Stuijk *et al.* 2010]. This section uses the M-JPEG decoder to: 1) exhibit the flexibility of our framework to capture dynamic behavior and allow customized simulations, and 2) compare the analysis results between a conservative analysis and an analysis taking into account dynamic behavior.

The decoder behavior described in Section 4.2 is firstly translated into our model composing of two behaviors: b_1 and b_2. We consider the platform that can be configured to support up to five processors $\{p_1, ..., p_5\}$ interconnected by an 2*3 Hermes NoC operating at $600MHz$. Each processor can operate on three frequencies $\{40MHz, 60MHz, 120MHz\}$,

and is associated with a memory buffering the data received. Processors p_1, p_2, p_3 are allocated to the three nodes in the first row, while p_4, p_5 are allocated on the first two nodes in the second row. Such a NoC takes around 6 cycles for a flip (32 bits) to traverse a hop (or node). The best and worst case transmission bandwidths between two nodes are computed by considering the shortest and longest routing. For example, the best case and worst case values between p_1 and p_2 are 3.2 bit/ns and 0.64 bit/ns. For each task, its execution time cost lower and upper bounds are obtained by taking the best and worst profiled values by using the SoCLib environment [SoClib 2012]. Lower and upper bounds of energy cost can also be obtained by using existing profiling tools, e.g., Sim-Panalyzer [sim 2013]. On average, the lower bound for task execution time (resp. energy) cost is around 10% less than the upper bound. Energy costs for all PEs when they are idle are set to 0.

First, we use our framework as a conservative analysis tool. In this case, the worst-case cost values for tasks and communication bandwidths are taken, and the behavior that has the worse costs is taken as the static application behavior. Second, we use our framework to capture the two application dynamic behavior, and task cost values and communication bandwidths are got by a random selection strategy from defined intervals. In both cases, we simulate the frame processing for 30 iterations. The two experiments have found respectively 56 and 83 Pareto solutions, both taking less than 10 minutes. Among them, 31 solutions have the same mapping and frequency values. Not surprisingly, the time and energy costs of these common solutions obtained in the second experiment are much better than the first experiment. Around 63% time and 17% energy cost reductions can be observed on average. Energy reduction is relatively smaller, as we do not take into account the energy cost for communications. Our framework is flexible enough to capture system dynamic behaviors, and support both conservative and customized designs.

4.7 Summary and Discussion

This chapter presented a high-level framework for the rapid and cost-effective design space exploration (DSE), devoted to the design of (adaptive) data-intensive applications on MPSoCs. A multi-clock modeling of both software and hardware has been considered by exploiting the notion of abstract clocks borrowed from synchronous dataflow languages. The implementation of the proposed DSE relies on a powerful evolutionary algorithm for heuristic exploration. Some very promising experimental results have been carried out on a few application examples on multiprocessor execution platforms. Our approach is an ideal complement to lower-level design assessment techniques for MPSoCs, such as physical prototyping and simulation. It also aims to serve as an intermediate reasoning support that is usable, from very high-level MPSoC models (e.g., in UML Marte profile), to deal with critical design decisions. Finally, the whole framework has been implemented in a freely available tool.

The presented framework can be used to deal with the first design issue, i.e., the design of a configuration of adaptive MPSoCs, by restricting the adaptive application behavior to a static application behavior or configuration. The DSE process is able to generate a set of implementation choices that provide a trade-off in resource usage, performance and energy/power consumption. They can be used by a run-time mechanism to adapt the mapping in different run-time situations. Furthermore, the framework is also flexible enough to capture the adaptive behaviors of MPSoCs. It can thus be used as a high level simulator for adaptive MPSoCs to evaluate customized run-time managers. These two features will be presented with more details in Chapter 6.

The static analysis of data-flow application designs with predictable behaviors, has

mainly based on data-flow models, such as Kahn Process Networks (KPNs) and Synchronous Data-Flows (SDFs). KPNs are quite expressive and can capture dynamic application behaviors. The expressiveness power, on the other hand, makes it difficult to predict their precise behavior over time. Existing techniques based on KPNs, such as [Sigdel 2011], [Erbas *et al.* 2007], [Schor *et al.* 2012] usually employ simple run-time scheduling strategies e.g., first come first served algorithms, and do not investigate the design of scheduling algorithms. In our framework, we study admissible scheduling requirements, and propose a correct by construction scheduling algorithm. The relative simple and static nature of SDFs [Stuijk *et al.* 2011] make it possible to develop design-time analysis techniques (e.g., [Stuijk *et al.* 2008] [Ghamarian *et al.* 2006]), as well as efficient scheduling strategies (e.g., [Stuijk 2007]). For adaptive embedded systems, however, SDFs are not expressive enough to capture their dynamic behaviors. Abstracting away the system dynamic behavior by using SDFs would overestimate resource requirements [Stuijk *et al.* 2010]. The Scenario-Aware Dataflow (SADF) MoC has thus been employed (e.g., [Stuijk *et al.* 2010], [Stuijk *et al.* 2011] to combine a finite state machine-like structure with SDFs to deal with the modeling and analysis of adaptive applications. Compared to these SDF and SADF based approaches, they do not investigate the impact of potential delay between processor cycles on scheduling.

Our approach enriches the vision of the usual application of the synchronous model [Benveniste *et al.* 2003], which abstracts away the system quantitative time properties by considering that a program is faster than its environment, by encoding the quantitative time via abstract clocks. The resulting model provides a uniform support for design assessment w.r.t. quantitative properties beyond those addressed usually with the synchronous model. The results presented in this chapter are follow-up works of [Abdallah 2011], [Abdallah *et al.* 2012] and [Gamatié 2012]. Compared to them, this chapter studies the admissible scheduling requirements, proposes a correct by construction scheduling algorithm, and deals with adaptive/dynamic system behaviors. A preliminary work of parts of the results has been presented in [An *et al.* 2012a] [An *et al.* 2012b]. Compared to it, the CLASSY framework now is able to capture adaptive/dynamic application behavior, deals with the communication aspect more elaborate, and reasons on elementary costs given as best and worst case values.

The major limitation of our approach is the supported application models: applications described as cyclic component graphs are not currently addressed. To deal with this, a memory operator should be introduced in our framework, and this is one of the future work. In addition, the storage distribution computed by our design space exploration framework for application mapping solutions on MPSoCs is not optimal. We are considering to combine our framework with [Stuijk *et al.* 2008] in order to address this issue. Third, the manual transformation from MARTE (or SDF, CSDF) described application and platform specifications to our reasoning framework is tedious, and can become very complex and error-prone if the problem becomes big. An automatic transformation is thus needed.

Chapter 5

Discrete Control for Reconfiguration Management of Adaptive MPSoCs

Contents

5.1 Motivation and Contribution **70**
5.2 Adaptive MPSoCs Implemented on FPGAs **71**
5.2.1 Hardware Architecture 71
5.2.2 Application Software 71
5.2.3 Task Implementation 72
5.2.4 System Reconfiguration 72
5.2.5 System Objectives 73
5.2.6 High Level Modeling of Adaptive MPSoCs 73
5.3 Modeling Reconfiguration Management Computation as a DCS Problem . **74**
5.3.1 Architecture Behaviour 74
5.3.2 Application Behaviour 74
5.3.3 Task Execution Behaviour 77
Models Neglecting Reconfiguration Overheads 78
Models Taking into Account Reconfiguration Overheads 78
5.3.4 Global System Behaviour Model 80
5.3.5 System Objectives 81
5.4 Automatic Manager Generation by Using BZR **81**
5.4.1 BZR Encoding of the System Model 82
5.4.2 Enforcing Logical Control Objectives 83
5.4.3 Enforcing Optimal Control Objectives 85
5.4.4 Enforcing the Combined Logical and Optimal Objectives 87
5.5 Experimental Results **87**
5.6 Case Study **89**
5.6.1 Case Study Description 89
5.6.2 Controller Generation and Integration 90
Controller Generation 90
Controller Integration 92
5.6.3 System Implementation 92
5.7 Summary and Discussion **93**

This chapter focuses on the safe reconfiguration management of adaptive MPSoCs. We advocate the application of the discrete control technique to address the design of correct reconfiguration managers. We focus on MPSoCs implemented on FPGA-based reconfigurable architectures, which can draw various benefits such as efficiency and flexibility.

The chapter is organized as follows: Section 5.1 exposes the motivations of the study. Section 5.2 presents informally the class of computing systems of interest. Section 5.3 presents our modeling and reconfiguration manager computing framework through an illustrative example. Section 5.4 employs the existing BZR tool to perform DCS and automatically generate a manager satisfying system requirements. Section 5.5 exposes the experimental results regarding the scalability of our approach. Section 5.6 presents an experimental validation of our proposal by implementing a real-life video processing system on a Xilinx FPGA platform. Section 5.7 concludes.

5.1 Motivation and Contribution

Computing systems based on reconfigurable architectures such as FPGAs can draw various benefits, such as adaptability and efficient acceleration of compute-intensive tasks [Santambrogio 2010]. We consider MPSoCs implemented on *dynamically partially reconfigurable* (DPR) FPGAs (Field Programmable Gate Arrays). Such architectures support partial reconfigurations, where only part of the FPGA surface is reconfigured and reconfigurations are performed at run-time. These characteristics make them suitable for addressing constraints on resources (e.g., re-using some areas for different functions of applications that can be partitioned into phases) by adapting resources to available parallelisms according to environment variations. However, the dynamic reconfiguration capability of such architectures further complicates their design, and requires more efforts to maintain such complex infrastructure.

Due to the relative novelty of the DPR technologies, the reconfiguration management is usually addressed by using manual encoding and analysis, which is tedious and error-prone [Gohringer *et al.* 2008]. Automatic techniques are required to better address this problem, with the foreseeable increase in complexity. The autonomic computing, introduced in Section 2.3.3 of Chapter 2, is one solution to address their reconfiguration management, though it has been seldom applied to such hardware systems.

Our contribution. We adopt the discrete control technique to design the autonomic *MAPE-K* (Monitor, Analyze, Plan, Execute, based on Knowledge) loop for the autonomic management of DPR FPGA based MPSoCs. We propose a systematic modelling framework, where system application behaviour, task implementations and executions, architecture reconfigurations and environment are modelled separately by using *automata*. We encode the computation of an autonomic manager as a (Discrete Controller Synthesis) DCS problem w.r.t. multiple constraints and objectives e.g., mutual exclusion of resource uses, power cost minimization. The existing BZR tool, which encapsulate the DCS synthesis tool Sigali, is employed to validate our models and perform automatic manager generation. Extensive experiments have been performed to evaluate the scalability of our framework. An experimental validation of our proposal by implementing a real-life video processing system on a Xilinx FPGA platform is also performed. The main results of this chapter have been presented in [An *et al.* 2013a] [An *et al.* 2013b] [An *et al.* 2013c].

5.2 Adaptive MPSoCs Implemented on FPGAs

We present informally the class of computing systems of interest through an illustrative example. They are inspired by the self-adaptive embedded systems in [Eustache & Diguet 2008]. However, we address the problem in a different and original way.

5.2.1 Hardware Architecture

We consider a multiprocessor architecture implemented on an FPGA (e.g., Xilinx Zynq device), which is composed of a general purpose processor $A0$ (e.g., ARM Cortex A9), and a reconfigurable area divided into four tiles: $A1$–$A4$ (see Figure 5.1). The communications between architecture components are achieved by a *Network-on-Chip* (NoC). Each processor and reconfigurable tile implements a NoC Interface (NI). A fixed dual port memory buffer is associated with each tile, which means that at most two tasks can simultaneously access data stored in the shared memory. Reconfigurable tiles can be combined and configured to implement and execute tasks by loading predefined bitstreams, such as tiles A1 and A2 of Figure 5.1.

The architecture is equipped with a battery supplying the platform with energy. Regarding power management, an unused reconfigurable tile Ai can be put into sleep mode with a *clock gated mechanism* such that it consumes a minimum static power.

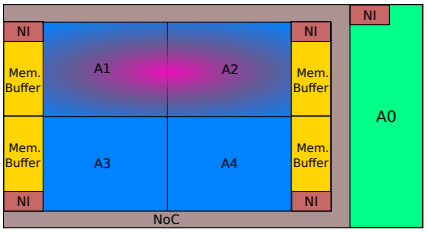


Figure 5.1: Architecture structure.

5.2.2 Application Software

We consider system functionality described as a *directed, acyclic task graph* (DAG). A DAG consists of a set of *nodes* representing the set of tasks to be executed, and a set of directed *edges* representing the precedence constraints between tasks. Note that we do not restrict the abstraction level of tasks associated with the nodes, and a task can be an atomic operation, or a coarse fragment of system functionality. Figure 5.2 shows an example consisting of four tasks.

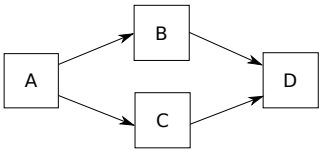


Figure 5.2: DAG application specification.

In our framework, unless otherwise specified, we suppose each task performs its computation with the following four control points:

- *being requested* or invoked;
- *being delayed*: requested but not yet executed;
- *being executed*: to be executed on the architecture;
- *notifying execution finish*, once it reaches its end.

Occurrences of control points *being requested* and *notifying finishes* depend on runtime situations, and are thus unpredictable and uncontrollable. The way of *delaying* and *executing* tasks is taken charge by a runtime manager aiming to achieve system objectives.

5.2.3 Task Implementation

Given a hardware architecture, a task can be implemented in various ways characterised by various parameters of interest, such as the set of used reconfigurable tiles (rs), worst case execution time (WCET) (wt), reconfiguration time (rt), and power peak pp. For instance, task A may have the two following implementations:

- Implementation 1: $rs_1 = \{A1\}$, $wt_1 = 200$, $rt_1 = 10$, $pp_1 = 180$;
- Implementation 2: $rs_2 = \{A3, A4\}$, $wt_2 = 100$, $rt_2 = 15$, $pp_2 = 250$;

Table 5.1 gives the considered implementations and corresponding profiled characteristics of tasks A, B, C, D.

	Implementations (resource set, WCET, reconfig. time, power peak)		
Tasks	Implementation 1	Implementation 2	Implementation 3
A	$(\{A1\}, 200, 10, 180)$	$(\{A3, A4\}, 100, 15, 250)$	-
B	$(\{A2\}, 450, 20, 120)$	$(\{A1, A2\}, 300, 25, 160)$	$(\{A1, A2, A3, A4\}, 150, 30, 400)$
C	$(\{A3\}, 240, 15, 100)$	$(\{A3, A4\}, 100, 20, 250)$	-
D	$(\{A1\}, 250, 12, 200)$	$(\{A1, A2\}, 100, 15, 350)$	$(\{A1, A2, A3, A4\}, 50, 20, 450)$

Table 5.1: Profiled task implementation characteristics for the working example.

Among the possible task implementations, a run-time manager is in charge of choosing the best implementation at run-time according to system objectives.

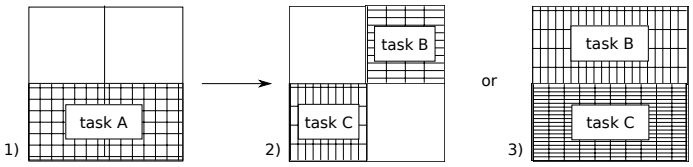


Figure 5.3: Configurations and reconfigurations.

5.2.4 System Reconfiguration

Figure 5.3 shows three system configuration examples. In configuration 1, task A is running on tiles A3 and A4 while tiles A and B are set to the sleep mode. Configurations 2 and 3 show two scenarios with tasks B and C running in parallel. Once task A finishes its execution according to the graph of Figure 5.2, the system can go to either configuration

2 or configuration 3 depending on the system requirements. For example, if the current state of the battery level is low, the system would choose configuration 2 as configuration 3 requires the complete FPGA working surface and therefore consumes more power. On the contrary, when the battery level is high, configuration 3 would be chosen if the user expects a better performance.

5.2.5 System Objectives

System objectives define the system functional and non-functional requirements. This section gives the objectives considered in the chapter, and categorises them as logical and optimal control objectives. Generally speaking, logical objectives concern state exclusions, whereas optimal objectives target the states associated with optimal costs.

Considered logical control objectives are as follows:

1. resource usage constraint: exclusive uses of reconfigurable tiles $A1$-$A4$;
2. dual accesses to the shared memory (i.e., at most two functions running in parallel);
3. energy reduction constraint: switch tiles to
   (a) sleep mode when executing no task;
   (b) active mode when needed;
4. reachability: system execution can always finish once started;
5. power peak constraint: power peak of hardware platform is constrained w.r.t battery levels;

Optimal control objectives of interest are as follows:

6. minimise power peak of hardware platform;
7. minimise WCET of system executions;
8. minimise worst case energy consumption of system executions.

5.2.6 High Level Modeling of Adaptive MPSoCs

The aforementioned informal system description can be satisfactorily modeled with high-level modeling formalisms such as MARTE [Object Management Group 2013a]. It offers a rich set of concepts to describe different features of adaptive embedded systems.

- Architecture behavior, which concerns the sleep and active mode switches of the reconfigurable tiles, can be modeled by using the UML Finite State Machines (FSMs).
- Application behavior, which is described by a directed task graph, can be modeled by using the *General Component Modeling* (GCM) package.
- Task implementation behavior, which concerns different task implementation modes or configurations and their transitions, can be captured by FSMs. Moreover, to associate characterized parameter values with the corresponding implementation configurations, the UML comments can be used.
- System reconfiguration behavior, which concerns different system configurations and their reconfigurations, can be modeled by the UML FSMs and comments as well.
- System objectives can be captured by using UML comments.

5.3 Modeling Reconfiguration Management Computation as a DCS Problem

We specify the modelling of the computing system behaviour and control in terms of labelled automata. System objectives are defined based on the models. We focus on the management of computations on the reconfigurable tiles and dedicate the processor area $A0$ exclusively to the resulting controller.

5.3.1 Architecture Behaviour

The architecture (see Figure 5.1) consists of a processor $A0$, four reconfigurable tiles $\{A1, A2, A3, A4\}$ and a battery. Each tile has two execution modes, and the mode switches are controllable. Figure 5.4(a) gives the model of the behaviour of tile Ai. The mode switch action between Sleep (*Sle*) and Active (*Act*) depends on the value of the Boolean controllable variable c_a_i. The output act_i represents its current mode.

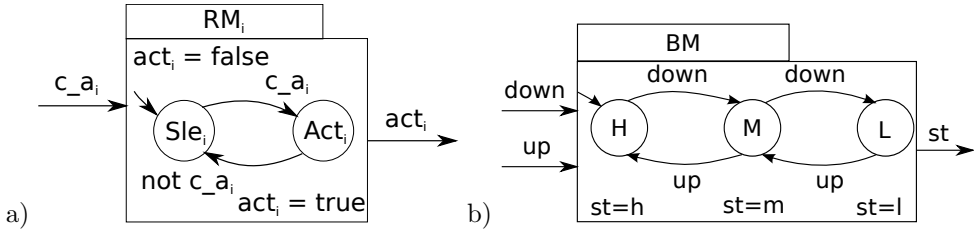


Figure 5.4: Models RM_i for tile Ai, and BM for battery.

The battery behaviour is captured by the automaton in Figure 5.4(b). It has three states labelled as follows: H (high), M (medium) and L (low). The model takes input from the battery sensor, which emits level *up* and *down* events, and keeps track of the current battery level through output *st*.

5.3.2 Application Behaviour

Software application is described as a DAG, which specifies the tasks to be executed and their execution sequences and parallelism. We capture its behaviour by defining a *scheduler automaton* representing all possible execution scenarios. It does so by keeping tracking of application execution states and emitting the *start* requests of tasks in reaction to the task *finish notifications*.

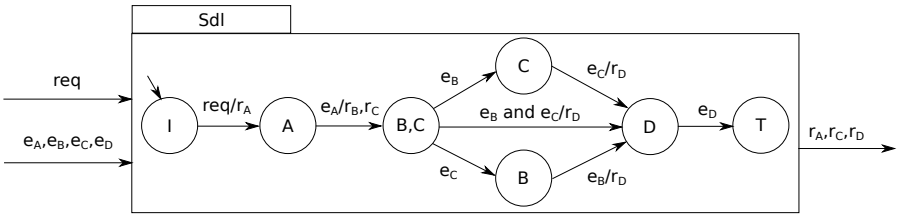


Figure 5.5: Scheduler automaton *Sdl* capturing application execution behaviours.

Figure 5.5 shows the scheduler automaton of the application in Figure 5.2. It starts the execution of the application by emitting event r_A, which requests the *start* of task A,

upon the receipt of application request event req in the idle state I. Upon the receipt of event e_A notifying the finish or *end* of A's execution, events r_B and r_C are emitted together to request the execution of tasks B and C in parallel. Task D is not requested until the execution of both B and C is finished, denoted by events e_B and e_C. It reaches the final state T, implying the end of the application execution, upon the receipt of event e_D.

In the following, we describe systematically how to obtain such a scheduler automaton from a DAG described application.

Scheduler Automaton Derivation. The scheduler automaton or LTS of a DAG described application captures the dynamic execution behavior of the application. Its states represent the tasks that are executing. They are denoted and labeled by the names of these tasks. It has an initial state I, i.e., the idle state, which means the application has not been invoked, and an end state T, which means that the application has finished its execution. The automaton input events are the task end events e_i and the application request event req, while its output events are the task request events r_i. Its transitions are of the form g/a, where g is a firing condition, and a is an action. A firing condition is a boolean expression of input events, and an action is a conjunction of output events. *Note*: 1) we suppose the application is only invoked once. If it is allowed to be repeatedly invoked, the end state would be the same to the initial state. 2) if the graph has a task that has more than one instance, the instances are then seen as different tasks by the algorithm.

Algorithm 3 illustrates how to construct the scheduler automaton for a DAG. It derives the automaton from initial state I to end state T by exploring the state space of the application execution w.r.t. the DAG.

- *Inputs:* a directed, acyclic task graph $< T, C >$, where T and C represent respectively the set of tasks, the set of edges.
- *Local variables and functions* used in the algorithm:
  - s, $nextState$: a state, with element $taskSet$ representing the set of tasks *associated* to the state (i.e., the tasks executing in the state);
  - drawState(s): a function that draws state s, labeled by $s.taskSet$;
  - drawTrans($source, sink, transition\ label$): a function to draw a transition from state $source$ to state $sink$ guarded by $transition\ label$;
  - drawnStates: the set of states that have been drawn out;
  - $stateQueue$: a FIFO queue, keeping track of the states to be processed, with function $popup()$ to return and delete the first state element, and function $add(s)$ to add state s to the end of the queue;
  - $t_i.prec$: the set of tasks that immediately precede task t_i;
  - $readyTaskSet$: the set of tasks that are enabled to execute;
  - tc: a set of tasks, or a task combination;
  - powerSet(set of tasks): the power set of the $set\ of\ tasks$ without $\emptyset$;
  - $traversed(s)$: a function that returns the set of states (in the drawn automaton so far) that are traversed by some path from state I to state s (states I and s included), with element $taskSet$ to return the union of the tasks associated with all its states.

At line 1, the initial state, i.e., idle state I is drawn denoted by $drawn(I)$. The set of drawn states $drawnStates$ is thus initialized to $\{I\}$ at line 2. State queue $stateQueue$ stores the states that have been drawn but not processed. It is initialized to have element I at line

Algorithm 3 Scheduler Automaton Derivation

```
1: drawState(I);
2: drawnStates = I;
3: stateQueue = stateQueue.add(I);
4: for all t_i ∈ T do
5:   if t_i.prec = ∅ then
6:     readyTaskSet = readyTaskSet ⋃ t_i;
7:   end if
8: end for
9: while stateQueue! = ∅ do
10:  s = stateQueue.popup();
11:  if s = I then
12:    nextState.taskSet = readyTaskSet;
13:    drawState(nextState);
14:    drawTrans(I, nextState, req/r_readyTaskSet);
15:    drawnStates = drawnStates ⋃ nextState;
16:    stateQueue.add(nextState);
17:  else if s = T then
18:    continue;
19:  else
20:    for all tc ∈ powerSet(s.taskSet) do
21:      readyTaskSet = ∅;
22:      for all t_i ∈ T− traversed(s).taskSet do
23:        if t_i.prec ⊆ (traversed(s).taskSet - s.taskSet) ⋃tc then
24:          readyTaskSet = readyTaskSet ⋃ t_i;
25:        end if
26:      end for
27:      nextState.taskSet = readyTaskSet ⋃ (s.taskSet - tc);
28:      if nextState.taskSet = ∅ then
29:        nextState = T;
30:      end if
31:      if nextState ∈ drawnStates then
32:        drawTrans(s, nextState, e_tc);
33:      else
34:        drawState(nextState);
35:        drawTrans(s, nextState, e_tc);
36:        drawnStates = drawnStates ⋃ nextState;
37:        stateQueue.add(nextState);
38:      end if
39:    end for
40:  end if
41: end while
```

3. Variable *readyTaskSet* represents the set of tasks that are *enabled* to execute once some event happens. A task is *enabled* if all its precedent tasks have finished their executions. Lines 4 to 8 set *readyTaskSet* to the set of tasks that have no precedent tasks, as such tasks can be executed immediately once the application is invoked/requested denoted by the receipt of event *req*. Lines 9 to 41 deal with the sequential processing of the states stored in *stateQueue*. The processing of a state concerns the drawing of its immediate following

states and the transitions, and put the new drawn states in the queue. The automaton derivation finishes when the queue becomes empty.

In the following, we describe how state s from queue *stateQueue* is processed. Line 10 evaluates the first state of the queue to s and removes it from the queue. Three types of states are distinguished and processed accordingly. They are initial state I, end state T and the rest. Lines 12 to 16 deal with the processing of idle state I. The algorithm firstly computes its following state *nextState* by evaluating the $readyTaskSet$ (got from Lines 4 to 8) to its *taskSet* at line 12. *nextState* represents the state once the application is invoked. Line 13 draws the state, and line 14 draws the transition from state I to *nextState* with label $req/\{r_i|t_i \in readyTaskSet\}$, where r_i is the request event of task t_i, denoted by $drawTrans(I, nextState, req/r_{readyTaskSet})$. Lines 15 and 16 add *nextState* to *drawnStates* and the end of *stateQueue*. If state s is the end state (line 17), the algorithm simply proceeds.

Lines 20 to 39 deal with the processing of state s that represents an application executing state between I and T. In general, the algorithm explores the finishes of all the possible subsets of the executing tasks in s (denoted by $s.taskSet$), and computes the following states accordingly. Given an element tc which represents a subset of the executing tasks in s, lines 21 to 38 deal with the drawing of the following states of s w.r.t. the simultaneous finishes of the executions of tasks in tc. Lines 21 to 26 compute the tasks that would become enabled if the set of tasks tc finishes. Variable $readyTaskSet$ initialized to $\emptyset$ at line 21 is used to keep these tasks. The union of the task sets associated with $traversed(s)$ denoted by $traversed(s).taskSet$ thus represents the tasks that have been executed before reaching s and are executing in current state s. At line 22, the algorithm explores the tasks that have not been requested (denoted by $T - traversed(s).taskSet$) to find out $readyTaskSet$ once tc finishes. Lines 23 to 25 decides whether t_i is enabled once tc finishes and adds t_i to $readyTaskSet$ if it is. t_i is enabled if the set of its precedent tasks is a subset of the union of tasks that have finished (denoted by $traversed(s) - s.taskSet$) and the tasks would finish denoted by tc. At line 27, *nextState* denotes the state following s due to the tasks in tc simultaneously finish. Its *taskSet* thus equals to the union of computed $readyTaskSet$ and the tasks that are still executing in s after tc finishes denoted by $s.taskSet - tc$. If $nextState.taskSet$ is $\emptyset$, this means that once the tasks in tc finish, no more task can become enabled or is still executing, i.e., all tasks have finished their executions and *nextState* is the end state T (lines 28 to 30). In the scheduler automaton, a state might have more than one precedent state. I.e., a state might have been drawn during the processing of some other precedent state. The algorithm thus checks, at line 31, if *nextState* has been drawn. If it has, only the transition needs to be drawn from s to *nextState* with label $\{e_{t_i}|t_i \in tc$ denoted by e_{tc} at line 32. Otherwise, the algorithm draws both state *nextState* and the transition at lines 34 and 35, and then updates *drawnStates* and *stateQueue* accordingly.

5.3.3 Task Execution Behaviour

Before executing a task on an FPGA, the task implementation (i.e., a bitstream) should be loaded to reconfigure the corresponding tiles of the FPGA. The reconfiguration operations inevitably involve some overheads regarding e.g., time and energy. Sometimes, these reconfiguration overheads can be neglected, if the reconfiguration overheads are small enough and neglecting them does not have an impact on the global design. In some other cases, however, these reconfiguration costs cannot be neglected and must be taken into account. In this section, we distinguish these two cases and model their behaviors respectively. In the rest of this chapter, the models neglecting reconfiguration overheads are considered.

Models Neglecting Reconfiguration Overheads

In consideration of the four control points of task executions (see Section 5.2.2), the execution behaviour of task A associated with two implementations (see Section 5.2.3) can be modelled as Figure 5.6. It features an initial *idle* state I_A, a *wait* state W_A, and two *executing* states X_A^1, X_A^2 corresponding to two implementations of task A. Controllable variables are integrated in the model to encode the controllable points: being delayed and executed. Upon the receipt of *start* request r_A, task A goes to either:

- *executing* state $X_A^i, i \in \{1, 2\}$ if the value of *controllable* variable c_i leading to X_A^i is *true*, or
- *wait* state W_A if delayed, i.e., the value of Boolean expression $c = \bigvee c_i, i \in \{1, 2\}$ is false.

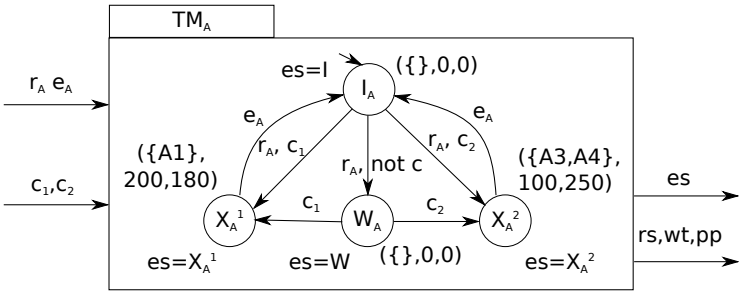


Figure 5.6: Execution behavior model TM_A of task A without considering reconfiguration time.

From wait state W_A, upon the receipt of event c_i, it goes to execution state X_A^i. When the execution of task A finishes, i.e., the finish or end notification event e_A is received, the automaton goes back to *idle* state I_A. Output es represents its execution state.

Local execution costs. The execution costs of different task implementations are different. As we neglect the reconfiguration overhead, i.e., reconfiguration time in the example, three cost parameters are considered here (see Section 5.2.3). We capture them by associating cost values denoted by a tuple (rs, wt, pp) with the states of task models, where: $rs \in 2^{RA}$ (RA is the set of architecture resources), $wt \in \mathbb{N}$ (a WCET value) and $pp \in \mathbb{N}$ (a power peak). The costs associated with *executing* states are the values associated with their corresponding implementations. For *idle* and *wait* states, apparently $rs = \emptyset, wt = 0, pp = 0$. Figure 5.6 gives the complete local model of task A.

Models Taking into Account Reconfiguration Overheads

To take into account reconfiguration overheads, the control point:

- *being kept loaded* or configured, after having finished,

should be added to the four task execution control points of Section 5.2.2. In this case, once a task finishes its execution, the controller should decide either keeping it loaded on the FPGA in order to avoid reconfiguration costs when it is executed again, or re-using the areas it occupies for other tasks. To simplify the modeling and better illustrate the modeling of reconfiguration overheads, in this section, we do not consider the control point *being delayed*: requested but not yet executed of Section 5.2.2, and still keep four control points for the task execution models.

We still take task A associated with two implementations (see Section 5.2.3) as an example to model its execution behavior. W.r.t. the four control points of task executions considered in this section, the execution behaviour of task A is modelled as in Figure 5.7. It features an initial *idle* state I, two *executing* and *loaded* state pairs (X_1, L_1) and (X_2, L_2) corresponding to the two different implementations. Each *loaded state* L_i corresponds to an *executing state* X_i, meaning that their corresponding task implementation has been configured in this state, and can be executed directly, i.e., go to state X_i, without the need to reconfigure. Controllable variables c_1, c_2 are integrated in the model to encode controllable points: *being executed* and *kept loaded*.

From idle state I, upon the receipt of *start* request r_A, task A goes to one *executing* state $X_i, i \in \{1, 2\}$ depending on the value of *controllable* variable c_i leading to X_i. L_i can only be reached from idle state I through its corresponding executing state X_i, meaning that the task has finished its execution in X_i (i.e., e_A is received), and remains configured on the corresponding occupied surface. From state L_i, once the task is requested again, it can either go to state X_i and be executed directly without being configured, or be configured in some new way and executed accordingly (i.e., go to state $X_j, i \neq j$). From state L_i, it can also go to state I if the controller at some moment decides to release its occupied surface for the use of other tasks. When the task ends, denoted by e_i, from an executing state X_i, it might go to state I as well, instead of state L_i, depending on the values of controllable variables, e.g., its occupied tiles are taken by other tasks.

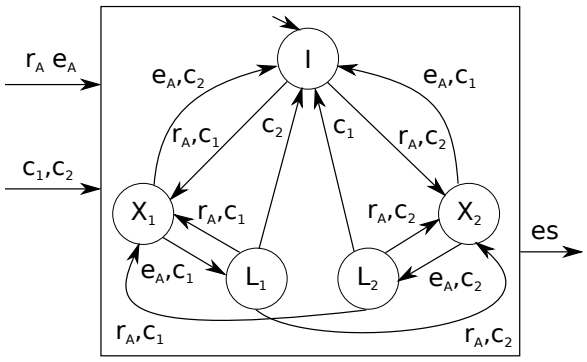


Figure 5.7: Execution model of task A taking into account reconfiguration overheads.

Local execution costs. The execution costs of different task implementations are different. Four cost parameters are considered (see Section 5.2.3). We capture them by associating cost functions denoted by a tuple (rs, wt, rt, pp) with the states of task models, where: $rs \in 2^{RA}$ (the occupied set of architecture resources), $wt \in \mathbb{N}$ (a WCET value), and $rt \in \mathbb{N}$ (a reconfiguration time). We now define the costs according to three types of states:

- idle state I: $rs = \emptyset, wt = 0, rt = 0, pp = 0$;
- loaded state L_i: $rs = rs_i, wt = 0, rt = 0, pp = 0$;
- executing state X_i: $rs = rs_i, wt = ec_i, rt = 0$ or $rt_i, pp = pp_i$.

rs_i represents the occupied resource set of the task implementation that corresponds to the loaded and executing state. rs has value rs_i for loaded states, as a task that is kept loaded still occupies the corresponding resources. The cost values of WCET wt and reconfiguration time rt for idle and loaded states are 0, as execution and reconfiguration time costs can

only be induced after a task is requested (i.e., reception of r). The cost values wt_i and rt_i corresponding to a task implementation should thus be evaluated to the corresponding cost functions of executing states. However, the reconfiguration cost of an executing state X_i is not always equal to the corresponding rt_i, but can also be 0 when the task has been loaded or configured. Therefore, regarding the task execution model in Figure 5.7, cost function rt of X_i should be defined in consideration of its previous states. It is evaluated to 0 if its previous state is L_i, i.e., at the moment the task is requested, it has been loaded. Otherwise, it is evaluated to rt_i, i.e., the task has to be loaded before executed. Take state X_1 in Figure 5.6(b) as an example, it has three incoming edges from states I, L_1 and L_2. The reconfiguration time cost rt is equal to 0 if it is reached from state L_1, and rt_1 if reached from I or L_2. W.r.t. the power peak function pp, no reconfiguration power peak cost is considered in the example, it is thus evaluated to pp_i in executing state X_i and 0 in the other states.

5.3.4 Global System Behaviour Model

The parallel composition of the control models for reconfigurable tiles RM_1-RM_4, battery BM and tasks TM_A-TM_D, plus scheduler Sdl comprises the system model:

$$\mathcal{S} = RM_1||...||RM_4||BM||TM_A||...||TM_D||Sdl$$

with initial state $q_0 = (Sle_1, ..., Sle_4, H, I_A, ..., I_D, I)$. It represents all the possible system execution behaviours in the absence of control (i.e., a run-time manager is not yet integrated). Each execution behaviour corresponds to a *complete path*, which starts from initial state q_0 and reaches one of the *final states*:

$$Q_f = (q(RM_1), ..., q(RM_4), q(BM), I_A, ..., I_D, T),$$

where $q(Id)$ denotes an arbitrary state of automaton Id.

Global costs. The costs defined locally in each task execution model need to be combined into global costs.

Costs on states. A system state q is a composition of local states (denoted by $q_1, ..., q_n$), and we define its cost from the local ones as follows:

- used resources: union of used resources associated with the local states, i.e., $rs(q) = \bigcup rs(q_i), 1 \leq i \leq n$;
- worst case execution time: this indicates how much time the system takes at most in this current state. It is thus defined as the minimal WCET of all executing tasks in this state, i.e., $wt(q) = min(wt(q_i), wt(q_i) \neq 0, 1 \leq i \leq n)$; Otherwise, if no task is executing in the state, i.e., $\forall 1 \leq i \leq n, wt(q_i) = 0$, $wt(q) = 0$;
- power peak: the sum of values associated with the local states, i.e., $pp(q) = \sum(pp(q_i)$, $1 \leq i \leq n)$;
- worst case energy consumption: the product of the worst case execution time and power peak of the system state, i.e., $we(q) = pp(q) * wt(q)$.

Costs on paths. We also need to define the costs associated with paths so as to capture the characteristics of system execution behaviours. Given path $p = q_i \rightarrow q_{i+1} \rightarrow ... \rightarrow q_{i+k}$, and costs associated with system states, we define corresponding costs on path p as follows:

- WCET: the sum of WCETs on the states along the path, i.e., $wt(p) = \sum wt(q_j), i \leq j \leq i + k$;

- power peak: the maximum value on the states along the path, i.e., $pp(p) = max(pp(q_j), i \leq j \leq i+k)$;
- worst case energy consumption: the sum of the worst case energy consumptions on the states along the path, i.e., $we(p) = \sum we(q_j), i \leq j \leq i+k$.

5.3.5 System Objectives

The two types of system objectives: logical and optimal ones, can then be described in terms of the states and the costs defined on the states or paths of the model.

Logical control objectives. For any system state q, we want to enforce the following:

(1) exclusive uses of reconfigurable tiles by tasks: $\forall q_i, q_j \in q, i \neq j$, that $rs(q_i) \bigcap rs(q_j) = \emptyset$;

(2) dual accesses to shared memory, i.e., at most two tasks can access the memory at the same time:
$\sum v_i \leq 2$, s.t. $v_i = \begin{cases} 1 & q_i \in X_i \\ 0 & \text{otherwise} \end{cases}$, where X_i represents the set of executing states of corresponding task;

(3.a) switch tile A_i to sleep mode, when executing no task: $\nexists q_i \in q, Ai \in rs(q_i) \Rightarrow act_i = false$;

(3.b) switch tile A_i to active mode when executing task(s): $\exists q_i \in q, Ai \in rs(q_i) \Rightarrow act_i = true$;

(4) reachability: Q_f is always reachable.

(5) battery-level constrained power peak (given threshold values P_0, P_1, P_2): $pp(q) < P_0$ (resp. P_1 and P_2) when battery level is high (resp. medium and low).

Optimal control objectives. Such objectives can be further classified into two types of objectives: one-step optimal and optimal control on path objectives. We use pseudo functions *max* and *min* in the following to represent the maximisation and minimisation objectives, respectively.

One-step optimal objectives. One-step optimal objectives aim to minimise or maximise costs associated with states and/or transitions in a single step [Marchand & Samaan 2000]. Objective 4 of Section 5.2 belongs to this type.

(6) minimise power peak *pp* in the next states of state q: $min(pp, q)$.

Optimal control on path objectives. Such objectives aim to drive the system from the current state to the target states Q_f at the best cost [Dumitrescu *et al.* 2010]. Objective 5 and 6 are such objectives.

(7) minimise remaining WCET wt from state q: $min(wt, q, Q_f)$;

(8) minimise remaining energy consumption we from q: $min(we, q, Q_f)$.

5.4 Automatic Manager Generation by Using BZR

Given the system graphical models and objectives of Section 5.3, this section describes the controller synthesis by using BZR and the DCS tool Sigali. Logical and optimal objectives are treated differently.

5.4.1 BZR Encoding of the System Model

The automaton encoding of the system components in Section 5.3 can be translated to the BZR textual encoding easily, as shown in Section 3.4 of Chapter 3. Here we only illustrate the BZR encoding of automaton TM_A of Figure 5.6 as follows. The rest models can be encoded similarly. Note that: to represent the used resources of task A, Boolean variables $a\_onA1, ..., a\_onA4$ are used to represent respectively that task A is using tile $A1, ..., A4$ or not in the corresponding states.

```
node taskModelA (ra,ea,c1,c2: bool)
returns (a_onA1,a_onA2,a_onA3,a_onA4: bool; wt, pp: int)
let
   automaton
      state I do
          a_onA1 = false; a_onA2 = false; a_onA3 = false; a_onA4 = false;
          wcet = 0; pp = 0;
      until ra & c1 then XA1
      | ra & c2 then XA2
      | ra then WA
      state WA do
          a_onA1 = false; a_onA2 = false; a_onA3 = false; a_onA4 = false;
          wcet = 0; pp = 0;
      until c1 then XA1
      | c2 then XA2
      state XA1 do
          a_onA1 = true; a_onA2 = false; a_onA3 = false; a_onA4 = false;
          wcet = 200; pp = 180;
      until ea then I
      state XA2 do
          a_onA1 = false; a_onA2 = false; a_onA3 = true; a_onA4 = true;
          wcet = 100; pp = 250;
      until ea then I
      end;
tel
```

The BZR encoding of the global system behaviour can be obtained by composing all these models. Finally, the costs on system states are defined as described in Section 5.3.4. We show in the following the BZR code of the global system model, which is structured in a node, named *glosys*.

```
node glosys (req,ea,eb,ec,ed,down,up: bool)
returns (a_onA1,a_onA2,a_onA3,a_onA4,b_onA1,b_onA2,b_onA3,b_onA4,
         c_onA1,c_onA2,c_onA3,c_onA4,d_onA1,d_onA2,d_onA3,d_onA4,
         act1,act2,act3,act4,battery_h,battery_l: bool;
         pp, wcet, we: int)
var
     ra, rb, rc, rd: bool;
     wt_a, pp_a, wt_b, pp_b, wt_c, pp_c, wt_d, pp_d:int;
let
 (* 4 tasks *)
 (a_onA1,a_onA2,a_onA3,a_onA4,wt_a, pp_a) = inlined taskModelA(ra,ea,c1,c2);
```

```
 (b_onA1,b_onA2,b_onA3,b_onA4,wt_b, pp_b) = inlined taskModelB(rb,eb,c3,c4);
 (c_onA1,c_onA2,c_onA3,c_onA4,wt_c, pp_c) = inlined taskModelC(rc,ec,c5,c6);
 (d_onA1,d_onA2,d_onA3,d_onA4,wt_d, pp_d) = inlined taskModelD(rd,ed,c7,c8);
 (* scheduler *)
 (ra,rb,rc,rd,target) = inlined scheduler(req,ea,eb,ec,ed);
 (* 4 reconfig. regions *)
 act1 = inlined region_manager(c_a1);
 act2 = inlined region_manager(c_a2);
 act3 = inlined region_manager(c_a3);
 act4 = inlined region_manager(c_a4);
 (* battery *)
 (battery_h,battery_l) = inlined battery(down,up);

 (* global costs: WCET, power peak and worst energy *)
 pp = pp_a + pp_b + pp_c + pp_d;
 wcet = if (wt_a <= 0 & wt_b <= 0 & wt_c <= 0 & wt_d <= 0) then 0
        else min(wt_a,wt_b,wt_c,wt_d);
 we = wcet*pp;
tel
```

The inputs of the system are the application request req, the notifications of task execution finishes ea, eb, ec, ed and the battery level transition signals $down, up$. The outputs represent correspondingly the resource usages of reconfigurable tiles, denoted by $t\_onAi, t = \{a, b, c, d\}, i = \{1, 2, 3, 4\}$, the states of the tiles, denoted by $acti, i = \{1, 2, 3, 4\}$, the states of battery levels, denoted by $battery\_h, battery\_l$ the power peak pp, WCET wt and worst energy consumption we of the current system state. The start requests ra, rb, rc, rd and the time and energy costs $wt_t, pp_t, t = \{a, b, c, d\}$ of the four tasks are defined as local variables by using the keyword var. In the body of the node, the system component models are composed by ";". The global costs of the system states are then defined based on the local models as described in Section 5.3.4. For the definition of WCET wt, $wt\_t <= 0, t = \{a, b, c, d\}$ means that task t is not active. The if clause thus means all tasks are in either idle or wait state. For simplicity, we use here the pseudo code $min(wt\_a, wt\_b, wt\_c, wt\_d)$ instead of many other if-else clauses to represent the minimal positive value of the four values.

5.4.2 Enforcing Logical Control Objectives

BZR *contracts* are able to directly encode the logical control objectives of Section 5.3 except for object 4) about reachability. This section focus on these objectives that can be encoded by *contract*, i.e., objectives 1, 2, 3 and 5. The following shows parts of the BZR *contract* defined based on the above global system model.

```
1  contract
2  var exclusive_tileA1,idle_tileA1,only_A_on_tileA1,...,only_D_on_tileA1,
    swt_sleep_tileA1,swt_act_tileA1,bound_pp: bool;
3  let
     (*exclusive usage of tile A1*)
4    exclusive_tileA1 = idle_tileA1 or only_A_on_tileA1 or
      ... or only_D_on_tileA1;
5    idle_tileA1 = not a_onA1 & not b_onA1 & not c_onA1 & not d_onA1;
6    only_A_on_tileA1 = a_onA1 & not b_onA1 & not c_onA1 & not d_onA1;
      ...
```

```
     (*switch tile A1 to sleep mode when running no task*)
7     swt_sleep_tileA1 = not idle_tileA1 or not act1;
     (*switch tile A1 to active mode when executing a task*)
8     swt_act_tileA1 = idle_tileA1 or act1;
     (*bounded power peak*)
9     bound_pp = if battery_high then (pp <= 500)
      else if battery_low then (pp <= 300)
      else (pp <= 400);
10 tel
11 enforce exclusive_tileA1 & swt_sleep_tileA1
     & swt_act_tileA1 & bound_pp
12 with (c_a1, c_a2, c_a3, c_a4, c_1,c_2, ... : bool)
```

Line 2 declares the local variables used within the contract with keyword *var*. They are declared as boolean variables by using keyword *bool*, and are defined in the body, i.e., Lines 4 to 9 between keywords *let* and *tel*. Variable $exclusive\_tileA1$ represents the exclusive usage of tile $A1$, i.e., objective 1, and is defined as the disjunction of five possible cases: tile $A1$ is idle denoted by $idle\_A1$; only one of the four tasks is running on tile $A1$ denoted by $only\_T\_on\_tileA1, T = \{A, B, C, D\}$. $idle\_A1$ and $only\_T\_on\_tileA1$ are defined at Lines 5 and 6 based on the states of the four task implementation models. t_onA1 represents that task t is using tile $A1$. objectives 3.a and 3.b Objectives 3.a and 3.b denoted by swt_sleep_tileA1 and swt_act_tileA1 are defined at Lines 7 and 8 by using the equivalent expressions of $idle\_A1 \Rightarrow not\ act1$ and $not\ idle\_A1 \Rightarrow act1$ respectively. $act1$ is the output of tile $A1$'s behavior model (see Figure 5.4). Line 9 defines the objective 5 denoted by $bound\_pp$. These objectives are then enforced at Line 11 with keyword $enforce$. All the relevant controllable variables to be computed are declared at Line 12 with keyword *with*. They are the controllable variables declared in the tile models (see Figure 5.4) and task implementation models (see Figure 5.6). The exclusive usages of the other tiles, their mode switch managements, and objective 2 can be encoded similarly.

Taking as input the system model and the contract, the BZR compiler can synthesize a controller (in C or Java code) automatically satisfying the defined objectives. There is also a graphical tool enabling the users to perform simulations of the controlled system by combines the controller with the system model.

Figure 5.8 shows a simulation scenario of the controlled system (see Table 5.1 for the implementation characteristics of the tasks) by using the graphical display tool sim2chro[1] developed by Yann Rémond. At instant 3, as labeled 1 in the figure, variables a_onA3 and a_onA4 become true, which implies that the second implementation of A which uses tiles $A3$ and $A4$ is chosen by the manager. At the same instant, tiles $A3$ and $A4$ are switched to the active mode, i.e., $act3, act4$ become true, which corresponds to objective 3.b). At instant 9, as shown by label 2 in the figure, task C finishes its execution by releasing tiles $A3$ and $A4$, i.e., c_onA3 and c_onA4 become false. At the same instant, tiles $A3$ and $A4$ are switched to the sleep mode, i.e., $act3, act4$ become false. This corresponds to objective 3.a). As shown in label 3, the system power peak pp is always less than 300, even though battery level is high. This is because that, firstly, the tasks cannot change its implementation once executed, and secondly, *down* and *up* events are uncontrollable. The power peak value is thus always kept under 300 to avoid the system goes to an invalid state where a task uses an implementation with a power peak bigger than the value that the lower level allows, i.e., 300, and the battery level goes low before it finishes. The exclusive usages of all the tiles can also be seen from the figure, e.g, for tile A1, the variables $t\_onA1, t = \{a, b, c, d\}$ do not

[1] http://www-verimag.imag.fr/~raymond/edu/distrib/

have value true at the same time during the simulation. The variable num_active_func representing the number of active tasks is always 1 during simulation. Objective 2) is thus also met.

5.4.3 Enforcing Optimal Control Objectives

So far, we have considered the objectives that can be directly encoded by a *contract* in BZR, i.e., objectives 1, 2, 3, 5. They are invariance objectives, which insure the invariance of a set of states of the system model. This is achieved by applying the Sigali controller synthesis operation *S_Security(S,prop)*, where S denotes the system model and *prop* represents the target subset of states. This operation has been encapsulated in the BZR compilation process, and thus can be used directly. Enforcing other control objectives, e.g., objective 4) ensuring a set of states reachable from initial states, and objective 6) driving the system go to the next states with optimal costs, however, needs other Sigali synthesis operations (see Section 3.3.2 of Chapter 3 for the description of these operations used in the section), and cannot be encoded by the BZR *contract* currently.

In the following, we describe how to combine BZR with Sigali to perform DCS regarding the other objectives. In this case, the BZR compiler serves as the front end to encode system behaviour, and produces an intermediate file with extension $z3z$, which is used as input by Sigali. The rechability and optimal control objectives can then be encoded and integrated into this file, which feeds the Sigali tool. The DCS is finally performed by Sigali to automatically generate a controller satisfying the objectives.

Reachability: objective 4). The reachability objective is to ensure that a set of target states E is always reachable. This has been implemented in Sigali by operation $S\_Reachable(S, prop)$, where S denotes the system model and *prop* represents the target subset of states. To apply this operation, the user need to open the generated z3z file, and replace the $S\_Security(S, prop)$ operation by $S\_Reachable(S, prop')$, where $prop'$ represents the target states. In our example, the target states are the final states defined in Section 5.3.4.

Optimal control within one step: objective 6). One-step optimal control is to drive the system to go to the next states optimizing the weights (or costs) associated with states. Sigali operation $Strictly\_Lower\_than(S, C, C\_Dup, duplicate\_states)$ is used for minimizing the power peak cost function (see Section 3.3.2 of Chapter 3). Here, we describe how to apply these operations with the BZR tool. There are three steps:

- firstly, compile the BZR program to generate the z3z file;
- secondly, duplicate the cost function and states of the system model, and invoke the corresponding synthesis operation in the generated z3z file;
- at last, perform the DCS operation by using the extended z3z file.

The tedious work in Step 2 has been realized in a dedicated tool which extends the z3z file generated by the BZR tool (available on demand).

Optimal synthesis on bounded paths: objectives 7) and 8). The objective for optimal control on paths aims to drive the system from the current state to the target states at the best costs. It aims to minimize the costs associated with states along paths. This has been implemented by the Sigali operation $S\_min\_weight\_path\_maxUC(S, C, T)$ (see Section 3.3.2 of Chapter 3). In our example, the targets states are the final states as defined in Section 5.3.4, and the cost functions are WCET, and worst case energy consumption, as defined in the BZR code in Section 5.4.1.

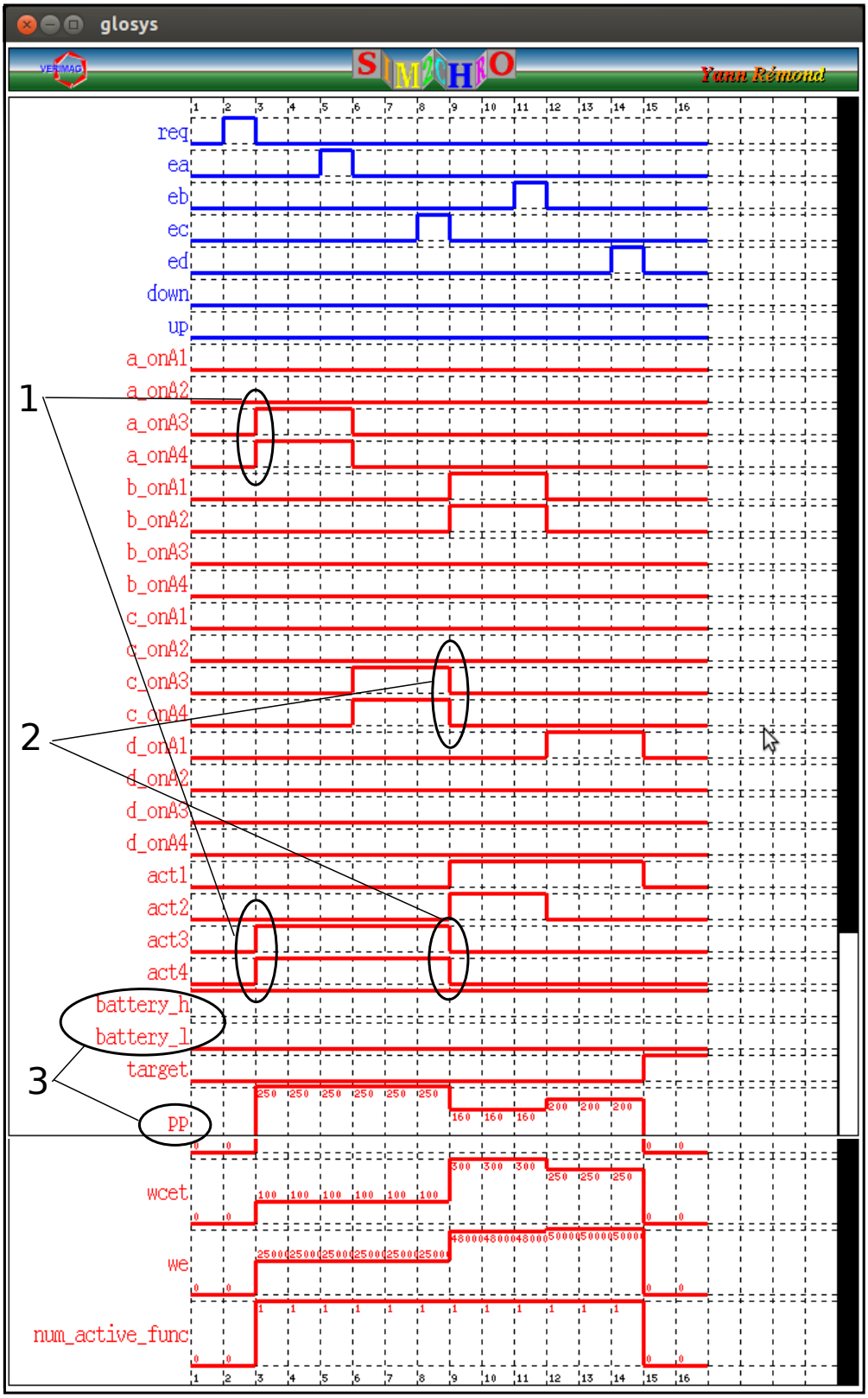


Figure 5.8: A simulation scenario.

5.4.4 Enforcing the Combined Logical and Optimal Objectives

The synthesis operations enforcing the logical and optimal control objectives can be combined. For instance, the synthesis operation for invariance *S_Security* and for one-step optimal *Strictly_Greater_than* can be combined to drive the system to always go to next optimal states within a set of states made invariant by the *S_Security* operation. The order of applying different operations does matter. Typically, the invariance synthesis should be applied before others (e.g., optimal control), such that it would not cut off (e.g., optimal) solutions if applied after. In our example, we enforce the invariance before the optimal ones, when both the two types of objectives are applied. The two optimal control synthesis operations are performed separately.

5.5 Experimental Results

We have carried on extensive experiments to evaluate the scalability of our framework. Table 5.2 shows our experimental results to compute the controllers by using the tools. It gives the time costs for different DCS operations corresponding to different system objectives w.r.t. different system models and state space sizes. The state space size of each system model is computed by simply multiplying state space sizes of its composed automata. The size of synthesized controllers varies from 50Kb (objective 2 on model 4:(2,3,2,3)) to 28Mb (objective 7 on model 6:(1^6)). We have started our experiments from the task graph of Figure 5.2. We then refine B to 3 tasks so as to increase the system model to 6 tasks, and at last, refine C to 3 tasks as well to address a 8 task model, as shown in Figure 5.9. We use

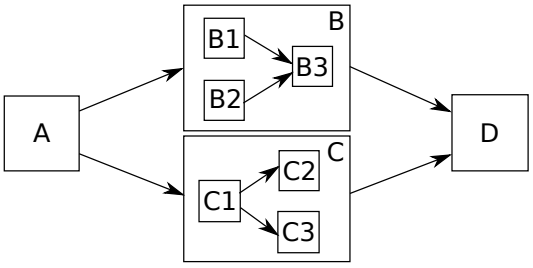


Figure 5.9: The refined task graph for experiments.

the notation $n : (m_1, ...m_n)$ to represent the models, where n denotes the number of tasks, and m_i the number of possible implementations of task i. Besides, we use m^k to represents k consecutive m's. E.g., $4 : (4^4)$ denotes $4 : (4, 4, 4, 4)$. All experiments are performed on a computer with a Intel(R) Core(TM)2 Duo CPU of 2.33GHz and a 3.8Gb main memory.

In our experiments, the DCS of invariance constraints, i.e., objectives 1-3,5, are applied directly on the original system model. On the basis of the resulting controller, the optimal and reachability ones are then performed. The objectives about invariance and reachability appear promising, while optimal ones are, unsurprisingly, explosive. An interesting point observed is that the time cost is not always increasing as state space size grows. System models consisting of more tasks but less possible implementations could have less synthesis times, e.g., DCS operations for 6:(1^6) model take less times than these for 4:(4^4) model. The reason may come from the fact that one-step-optimal and optimal-on-path syntheses require to examine the costs of next states, while more implementations of functions mean more choices to explore. Due to time and resource limitations, we have decided to stop the synthesis processes if not finished after 3 days of computation.

system model & **state space size** / target objectives	4:(2,3,2,3) **241,920**	4:(4^4) **806,736**	6:(1^6) **5,898,240**	6:(3,1,1,2,1,3) **16,588,800**	6:(3,2^4,3) **32,400,000**	8:(1^8) **566,231,040**	8:(3^3,2^5) **5,832,000,000**
1) exclusive usage of A_1-A_4	0.29sec	0.65sec	0.16sec	1.16sec	8.20sec	1.10sec	16min1sec
2) dual access to memory	0.12sec	0.49sec	0.10sec	0.69sec	1.50sec	1.29sec	23.05sec
3.a) switch to active mode	0.74sec	2.28sec	0.46sec	1.88sec	1min19sec	1.30sec	27min58sec
3.b) switch to sleep mode	0.76sec	2.01sec	0.22sec	1.74sec	2min11sec	0.90sec	41min29sec
5) battery-level constrained p.p.	0.89sec	2.23sec	0.68sec	4.18sec	21.18sec	2.21sec	49min24sec
4) reachability:	1.78sec	3.48sec	4.74sec	17.33sec	2min	25.16sec	3hr34min
6) minimize p.p. in next states	3min54sec	3hr18min	29min45sec	*stopped*	*stopped*	*stopped*	*stopped*
7)* minimize remaining WCET:	9min17sec	2hr43min	21min19sec	*stopped*	*stopped*	*stopped*	*stopped*

Table 5.2: The time costs for DCS operations corresponding to different target objectives w.r.t. different system models and state space sizes. $n : (m_1, ...m_n)$ denotes a model of n functions, with m_i denoting the number of possible implementations of function Fi. $*$ Objectives 7 and 8 are the same kind of operation (objective 8 is thus omitted here).

5.6 Case Study

In this section we describe an experimental case study and demonstrator, where some of the previous control models and objectives are applied to a concrete FPGA based platform.

5.6.1 Case Study Description

We consider a video processing system to be implemented on a platform containing an FPGA, so that the partial reconfigurations of the FPGA controlled by a synthesized controlled can be visualised. The processing system (see Figure 5.10) consists of a camera capturing images and sending them to the platform, a dispatcher feeding image pixels to the FPGA, a compositor aggregating pixels produced by the FPGA, and a screen displaying the processed images. Each captured image is divided into 9 areas by the dispatcher (see Figure 5.11), and the processing of each area is taken care by one dedicated reconfigurable tile of the FPGA. The FPGA is also divided into nine tiles (the same way as we had four in Figure 5.1). In this way, when a tile is reconfigured, one can see it on the screen. We consider three filtering algorithms (namely red, green and blue ones) that can be implemented on each reconfigurable tile to process images. When configured to process the same image, they have different performance values regarding some characteristics such as power peak, execution time. In the study, we suppose the power peaks of each tile for running the red, blue and green filters are respectively 3, 2 and 1.

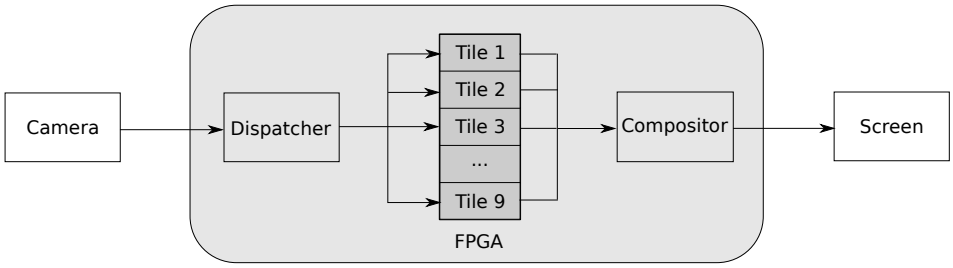


Figure 5.10: The video processing system case study.

Figure 5.11: Each image is divided into 9 areas for processing, with those covered by grids called corner areas, and the rest ones called cross areas.

The processing system can work at two different modes: *high* and *low*, controlled by the user through a switch on the platform. The user can also demand the use of the red filters for the processing of the four corner areas of images by using another switch on the platform. Apart from the user demands, the system also needs to respect the following three rules:

1. the four corner areas of the images to be displayed are of the same color, the five cross areas are of the same color, and the color of the four corner areas is different from the color of the five cross areas;

2. the global power peaks of the platform are bounded by 30 (respectively 20) in the *high* (respectively *low*) mode;

3. minimizing the power peaks of the next states.

A run-time manager is thus required to configure each reconfigurable tile of the FPGA by using one of the three filtering algorithms to filter images in the way satisfying the aforementioned requirements.

5.6.2 Controller Generation and Integration

We firstly follow the design flow in Section 5.3 to generate the run-time manager. We then describe how to integrate the BZR generated controller into the system implementation.

Controller Generation

We model the system reconfiguration behavior by using *synchronous parallel automata*, and DCS is then performed to generate a controller by using BZR.

System Modeling. Once the system gets started (modeled as the emission of event s), the controller should decide on the system initial state and configure the nice reconfigurable tiles of the FPGA accordingly. The behaviors of the two switches, denoted by $ModeSwitch$ and $CornerColorSwitch$, are captured by two boolean variables ms and gr respectively, with value *true* means switch on and *false* means switch off. When the value of ms is *true*, the user demands system to execute in *high* mode. When the value of gr is *true*, the user demands to use the red filters for processing the four corner areas of images.

The reconfiguration behavior of the system execution mode is captured by a three-state automaton (see Figure 5.12). Initially, it is in idle state I, once the system gets started denoted by s, it goes to either state High or Low depends on the value of ms. Boolean output h represents whether it is in mode *high*.

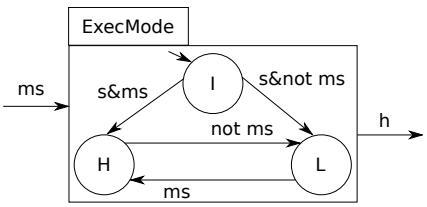


Figure 5.12: The model of system execution mode behavior, with Boolean input ms capturing $ModeSwitch$, and Boolean output h representing whether it is in mode *high*.

As the colors of the four corner areas are required to be the same, they always need the same filtering algorithm. We thus use one single automaton (see Figure 5.13) to model their choices among the three filtering algorithms. At the beginning, it is in state I. Once the system gets started, i.e., event s is received, it goes to state R, G or B, meaning that the red, green or blue filtering algorithm is used for processing the four corner areas of images, depending on the values of controllable variables $c1, c2, c3$. As running the red filter in a reconfigurable tile has cost 3, and R represents that all the four corner areas

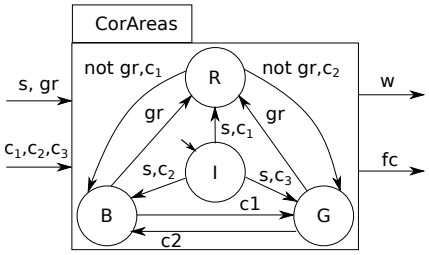


Figure 5.13: The model for choosing filtering algorithms for processing the four corner areas, where inputs $s \in \{true, false\}$ represents that the system gets started or not, $gr \in \{1, 0\}$ represents that the user switches on or off the corner color switch, and outputs $fc \in \{corR, corB, corG, corI\}$ represents the current state and $w \in \{12, 8, 4, 0\}$ represents the weight associated with the state.

run the red filter, we associated state R with cost 12. The same to the costs of states G and B. The automaton goes to state R upon the reception of event gr (i.e., the user switches on $CornerColorSwitch$), when it is in states G or B. The rest of the transitions (e.g., between G and B) are managed by the controller by evaluating controllable variables $c1, c2, c3$ according to system requirements.

The modeling for choosing the filtering algorithms for processing the five cross areas is done similarly. The main difference is that the user has control over using the red filter for processing the four corner areas by switching on switch $CornerColorSwitch$, but has such control for the five cross areas. The choice among the filters are always controllable. Figure 5.14 shows the model.

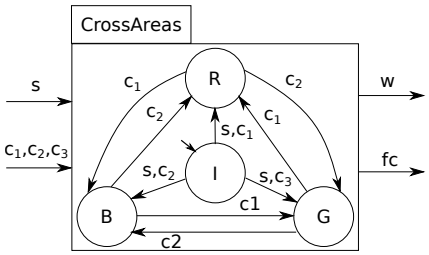


Figure 5.14: The model for choosing filtering algorithms for processing the five cross areas, where outputs $fc \in \{croR, croB, croG, croI\}$ represents the current state, and $w \in \{15, 10, 5, 0\}$ represents the weight associated with the state.

At last, all the aforementioned models are composed to derive the global system behavior. All the controllable variables define the control (i.e., reconfiguration) points that a controller should control according to the control rules. We employ BZR to automatically synthesize a controller satisfying the control rules. Firstly, the system models are encoded in BZR. This can be done easily as shown in Section 5.4.1. Secondly, the three control rules described in Section 5.6.1 should be specified separately. The control rules 1, 2 can be encoded in a BZR *contract* directly as given below. Variable *total_w* represents the global weight of the system, which is defined as the sum of the weights w in the four corner model and the five cross areas model. Regarding the optimal control objective, i.e., control rule 3, the generated z3z code needs to be modified by adding the corresponding optimal Sigali operation *Strictly_Lower_than*, as described in Section 5.4.3.

```
contract
```

```
var bound, exclusive:bool;
let
    bound = if(h) then (total_w <= 30) else (total_w <= 20);
    exclusive = not ((corR & croR) or (corG & croG) or (corB & croB));
tel
assume true
enforce bound & exclusive
with (c1,c2,c3, ... :bool)
```

Controller Integration

By feeding the BZR program to the BZR compiler, it generated the C code of the run-time manager within 5 seconds. The manager then needs to be integrated with the system. This section describes the structure of the BZR generated code and the way to integrate it with the system implementation.

The manager code is composed of two C code folders with overall size 77.8 kilobytes.One folder contains the C code of the generated controller which computes the values of controllable variables according to system states and inputs; the other folder contains the code for keeping track of the system states by performing state transitions according to system inputs, states and the values of computed controllable variables (this is done by invoking the functions in the former folder. Two additional C files named `main.c` and `main.h` are also generated by the compiler for simulation purpose. However, they can be easily adapted to serve as the interface between the manager and the system implementation. Next we take a closer look at the `main.c` code, and then describe the way of adapting them such that the manager code can be combined with the system.

The code of `main.c` can be divided into three parts:

- input part: system input variables declared in the system model;
- system model declaration and initialisation part: state variables and output variables (named as `mem` and `res` respectively) declaration and system state initialisation (by `reset(&mem)`);
- system states tracking and transition part: it is an infinite loop, and each iteration consists of the input variable evaluation, a step function: `sys_step(`*inputs*`, &mem)` computing output `res` and updating system state `&mem` according to the inputs and current state `&mem`, and the printout of output variable values.

To integrate the manager with the system, one needs to

1. pass the system input values to the input variables of the manager,
2. define within the infinite loop the timing to invoke the step function, and
3. interpret the output variables as system (reconfiguration) actions.

In the next section, we show how the generated manager can be integrated with a Xilinx platform.

5.6.3 System Implementation

We have implemented the video processing system on an ML605 board from Xilinx. It includes a Virtex-6 FPGA (XC6VLX240T), several I/O interfaces like switches, buttons, Compact Flash reader, and an external 512MB DDR3 memory. An Avnet extension card

(DVI I/O FMC Module) with 2 HDMI connectors (In and Out) has been plugged onto the platform so that it can receive and send video streams through the connectors.

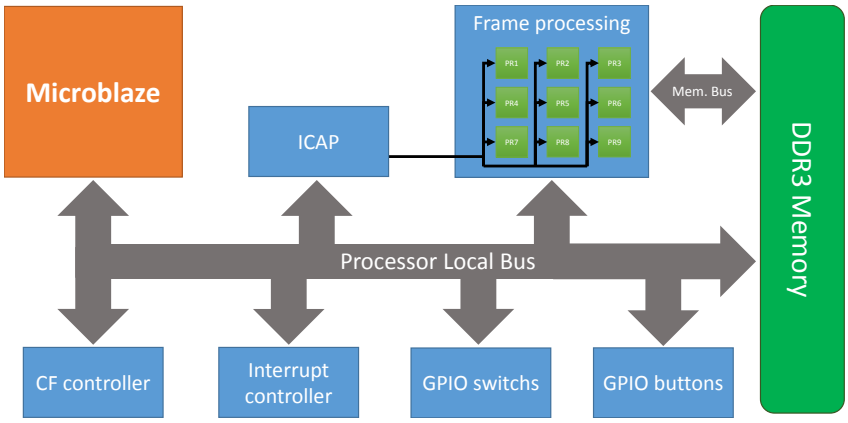


Figure 5.15: Global structure of the implementation

Figure 5.15 illustrates the global structure of our implementation. We have divided the FPGA surface into two regions: static and reconfigurable regions. Nine independent reconfigurable tiles are specified in the reconfigurable region. The tiles are in charge of the video processing tasks described in Section 5.6.1. The microblaze is a 32-bit soft-core processor synthesised on the static region of the FPGA (like $A0$ in Figure 5.1). It executes two main system tasks: the computed manager and the management of the configuration bitstreams. The latter task involves the control of related peripherals (i.e., Compact Flash memory, I/O interrupts, DDR3, ICAP) through corresponding implemented controllers. The external DDR 3 memory is used to buffer the frame pixel data of video streams, and store the software executable to be launched by the microblaze. The accesses to the DDR by the FPGA are managed by the DDR controller. We use a compact Flash card to store the bitstreams of the filter implementations on each reconfigurable tile. The C code of the manager is deployed on the microblaze as an infinite loop. It is invoked whenever the microblaze is interrupted. Two additional interrupt controllers (GPIO switches and GPIO buttons) are added for the platform to generate interrupts. They monitor the states of the buttons and switches, and generate interrupts when these states change. Once the manager is invoked, it is able to read the states of the switches and computes out a new configuration. The microblaze then selects the appropriate bitstreams from the Compact Card, and sends them to the ICAP to reconfigure the nice reconfigurable tiles.

Experimental results show that the integrated manager did meet the system requirements mentioned in Section 5.6.1 after a considerable number of tests.

5.7 Summary and Discussion

This chapter presented a general framework, based on a tool-supported synchronous variant [Marchand & Samaan 2000] of the *discrete control* [Ramadge & Wonham 1989] for the autonomic management of adaptive MPSoCs. We have focused on adaptive MPSoCs implemented on reconfigurable architectures especially DPR FPGAs. Such architectures constitute a platform for adaptive computing that is gaining widespread use. The contribution of the work is manifold: *i*) we propose a systematic modeling framework for DPR FPGA based embedded systems, where application behaviour (defined by a task graph), task im-

plementations and executions (characterized by parameters of interest e.g., time and power consumptions), architecture resource uses and reconfigurations are modeled separately by using automata; *ii*) we apply formalisms and tools from *discrete control*, supported by a programming language BZR and synthesis tool Sigali, to encode and perform the computation of an autonomic manager as a DCS problem; *iii*) we have performed extensive experiments to evaluate the scalability of our approaches, and an experimental validation of our proposal by implementing a video processing system on a Xilinx FPGA platform. Although focusing on adaptive MPSoCs implemented on reconfigurable architectures, our framework is also general enough to deal with the reconfiguration control of other adaptive MPSoCs. Its application to general MPSoCs will be illustrated by using an example in Chapter 6.

Compared to other existing techniques for self-management of adaptive systems, e.g., heuristics and machine learning techniques, the main advantage of control techniques is that they are able to provide formal correctness and/or performance guarantees. The authors in [Maggio *et al.* 2012] discuss some existing approaches applying standard control techniques such as Proportional Integral and Derivative controllers or Petri nets. However, discrete control has only been seldom applied [Guillet *et al.* 2012], and to the best of our knowledge, only few works have targeted at computing systems on reconfigurable architectures. In [Sironi *et al.* 2010], a reconfigurable architecture based evolvable system exploiting self-adaptive techniques is proposed. It is one of the first implementations of a FPGA-based self-aware adaptive system. It adopts the application heartbeats as monitoring framework, and a heuristic mechanism to switch between configurations. Self-management in the form of self-healing exploiting FPGAs is proposed in [Jovanović *et al.* 2008]. However, the approach does not involve control. A new architectural proposal in [Majer *et al.* 2007] provides a slot-based organisation of the reconfigurable hardware and an elaborate communication framework with good reconfiguration support. The focus is, however, on infrastructure aspects rather than on control.

Our approach is closer to that in [Eustache & Diguet 2008], but we focus on logical aspects and discrete control. In [Quadri *et al.* 2010b], a design flow, from high level models to automatic code generation, for the implementation of reconfigurable FPGA based SoCs is proposed. The system control aspects need to be modeled manually and integrated into the flow, while we advocate automatic controller synthesis. Compared to [Guillet *et al.* 2012], we have applied more elaborate DCS algorithms, and the integration into a design flow and compilation chain is more developed.

The major concern of our approach is the scalability issue, which is common to other formal techniques like model checking. An interesting means is to exploit the modular synthesis and compilation [Delaval *et al.* 2010], which allows the users to decompose big problems in a way that breaks down the combinatorial complexity. It is also interesting for specification and model structuring purposes. Another perspective is to enrich our models by taking into account other aspects, such as communication and memory access costs. At last, the automatic transformation from the MARTE described system specification to the automata in our framework, though seeming quite direct, has not been developed. Inspirations from [Guillet 2012] can be taken to deal with this.

Chapter 6

Combining Configuration and Reconfiguration Designs for Adaptive MPSoCs

Contents

6.1 Motivation and Contribution **95**
6.2 A Design Flow for Adaptive MPSoCs **96**
6.2.1 The Design Flow 96
6.2.2 Case Study: Continuous Multimedia Player 97
Informal Description 97
Step 1: MARTE Modeling 98
Step 2: Configuration Design 99
Step 3: Reconfiguration Manager Design 102
Step 4: Analysis Result Integration 104
6.3 CLASSY for the Design and Evaluation of Run-Time Managers **105**
6.3.1 An Illustrative Example 105
6.3.2 Run-Time Manager Design 106
Modeling Reconfiguration Behavior by Automata 106
BZR Encoding and Controller Generation 108
6.3.3 Simulation and Analysis of the Designed Manager by using CLASSY 108
6.4 Summary and Discussion **111**

6.1 Motivation and Contribution

We have presented a high level approach to deal with configuration analysis in Chapter 4, and an approach applying discrete control synthesis to deal with reconfiguration management in Chapter 5 of adaptive MPSoCs. The two approaches can thus be combined to form a complete design flow for the safe design of adaptive MPSoCs. Furthermore, the CLASSY framework presented in Chapter 4 is flexible enough to capture adaptive behavior, and allows the designers to design and analyze customized run-time managers by integrating the managers into its simulation process. This feature can be illustrated by applying the discrete control technique to design run-time managers that guide the CLASSY simulations.

This chapter is organized as follows: Section 6.2 presents the complete design flow for adaptive MPSoCs. Section 6.3 presents how the CLASSY serves as a simulation framework for the evaluation of designed managers. Section 6.4 concludes.

6.2 A Design Flow for Adaptive MPSoCs

Adaptive systems are able to adapt their behaviors dynamically in reaction to the run-time situations. The set of possible system behaviors can be seen as a set of system configurations, and the adaptation behavior can be seen as the reconfiguration between the configurations. Three possible adaptive aspects have been identified in Section 2.3.1 of Chapter 2, namely, application, platform and mapping adaptations. We, accordingly, define a *system configuration* as a composition of

- an *application configuration*, which is a specific application scenario,
- a *platform configuration*, which is an architecture instance, and
- a *mapping configuration*, which is the binding and scheduling of application tasks on the platform resources w.r.t. an application configuration and a platform configuration.

The *system reconfiguration* is, therefore, induced by reconfiguration at the three levels.

The adaptivity feature further complicates the design of MPSoCs, which leads to a real challenge about cost-effective and safe design methodologies of adaptive MPSoCs. Firstly, design correctness must be addressed in every possible configuration to ensure system reliability. This requires that the proper mapping and platform configuration solutions must be found for each application configuration w.r.t. system functional and non-functional requirements. Secondly, reconfiguration correctness must also be established to safely control the adaptation between system configurations. This requires a run-time manager, which monitors the system run-time situation and performs corresponding reconfiguration at the three levels if needed.

This section presents a design flow for the safe design of adaptive MPSoCs. We adopt the UML profile MARTE for the system modeling. The resulting models are transformed into formal models (i.e., abstract clocks, automata/BZR programs), by means of which, two levels of designs are carried out: the design for each application configuration w.r.t. system functional and nonfunctional properties by applying the CLASSY framework in Chapter 4, and the design of a correct controller for system reconfiguration by using discrete controller synthesis as in 5. At last, the design solutions are integrated into the original modeling framework. With the resulting models, existing hardware/software co-design framework such as Gaspard2 can then be used to automatically generate the corresponding programs, such as SystemC, and VHDL, for low level simulation and hardware synthesis. The main results of this section have been presented in [An *et al.* 2011].

Section 6.2.1 presents the design flow. Section 6.2.2 uses a case study to illustrate our approach.

6.2.1 The Design Flow

We present a design flow for the safe design of adaptive MPSoCs. Safety in the design is approached by using formal models to derive the design of each configuration, and generate a correct controller triggering reconfiguration. It goes according to the following steps:

- **Step 1: system modeling using the MARTE profile.** The UML standard profile MARTE [Object Management Group 2013a] is used to model considered adaptive systems. The system modeling concerns the modeling of the software application and hardware platform configurations, and their reconfiguration behavior. The resulting system models are then used for the two design issues: *design of each system*

configuration, and *derivation of a reconfiguration controller* that enforces the system requirements.

- **Step 2: system configuration design.** The design of each system configuration concerns for each application configuration computing a platform configuration and a mapping configuration such that system requirements regarding functional and non-functional ones are met. We use CLASSY to address this. In particular, it produces a set of Pareto-optimal design solutions regarding a trade-off of resource use, performance and energy consumption. Among the solutions, the designer can pick several candidates for each considered application configuration. They will be used by the run-time manager to decide the choice among these candidates depending on the run-time situations.

- **Step 3: system reconfiguration manager derivation.**

  We consider DCS, as used in Chapter 5, to deal with the design of a manager that controls the system reconfiguration w.r.t. run-time situations. The reconfiguration from all the three aspects, i.e., application, platform and mapping reconfiguration, should be taken into account. The application and platform reconfiguration models are specified by MARTE in Step 1. The resulting mapping configurations for each application configuration from Step 2 need to be organized and modeled by FSM or automata so that the mapping reconfiguration behavior can also be explored and managed in this step. These models are transformed into BZR programs, and the BZR, as shown in Section 3.4 of Chapter 3, is then used to automatically synthesize a correct manager satisfying system requirements.

- **Step 4: integration of design analysis results with the original system.**

  Finally, the design analysis results from Steps 2 and 3, i.e., the platform and mapping candidates for each application configuration, and the design manager, are integrated into the original modeling framework. With the models, existing hardware/software co-design framework such as Gaspard2 can then be used to perform further design and analysis.

6.2.2 Case Study: Continuous Multimedia Player

Informal Description

We consider a simple continuous multimedia (CM) player system (as modeled in Figure 6.1) extracted from [Rowe & Smith 1993]. The CM server (i.e., the media synchronization component in Figure 6.1) takes as input the streams of video and audio (CM) data packets from corresponding CM sources, synchronizes and assembles data from several packets into synchronized playable units, calculates the system time at which frames should be played, and dispatches them to output devices (e.g., the screener to play video and the speaker to play audio). The CM sources are responsible to convert the input media flow into required data format for further processing. The video stream is composed of a sequence of JPEG frames whereas the audio stream is captured in the form of Sparc audio.

Different temporal relations between media objects can be defined to achieve different functionality. Here, we suppose that the synchronization component has two modes or algorithms dealing with two different user requirements. They are taken and adjusted from [Blakowski & Steinmetz 1996], as shown in Figures 6.2 and 6.3. Both of these specifications are specified by using the reference point synchronization model. Each block of the media is

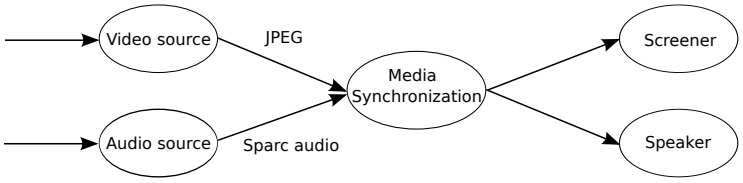


Figure 6.1: The continuous multimedia (CM) player system

a *logical data unit* (LDU) (e.g., frames for digital video) defined by the designer. Specially, a LDU is seen as a logical processing unit for media sources.

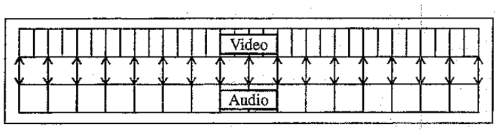


Figure 6.2: The lip synchronization specification.

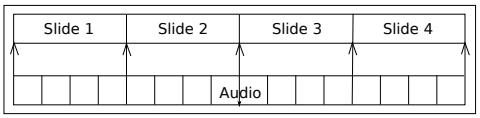


Figure 6.3: The slide show synchronization specification.

As we can see from Figure 6.2 (resp. 6.3), the lip synchronization mode (resp. slide show synchronization mode) each time takes an audio unit and two video units (resp. one slide and four audio units) for processing and assembling a single playable unit.

We suppose that the CM player is powered by a battery, which has two levels: high and not high. The users can switch between the two synchronization modes. Moreover, they expect the best performance, as long as the battery level allows.

Step 1: MARTE Modeling

Modeling of system functionality. With two different execution modes for the synchronization component, we have two functional or application configurations namely *LipSync*, as modeled by MARTE in Figure 6.4, and *SlideShowSync*. The stereotype ≪*configuration*≫ is used to represent a specific configuration with the name labeled below, which is associated with the value of the *mode* noted on the top right to indicate when the configuration is active.

Reconfiguration modeling. Having a number of possible configurations/modes for the application functionality, the UML *Finite State Machine (FSM)* is used to model and manage all these configurations as well as their switches. As shown in Figure 6.5, the FSM specifies the application reconfiguration behavior. It has two states corresponding to the two configurations we have mentioned above. Each state is associated with a specific mode value and the active configuration is the one having the same mode value of the current state of the FSM.

Modeling of execution platform. For the execution platform of this simple CM player system, we consider a hardware architecture composed of two processors, a hardware accelerator and memory devices adapted from the model in [Quadri *et al.* 2010a] (see Figure

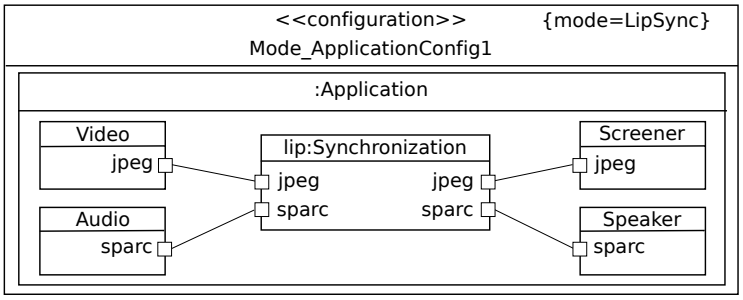


Figure 6.4: An application configuration: LipSync.

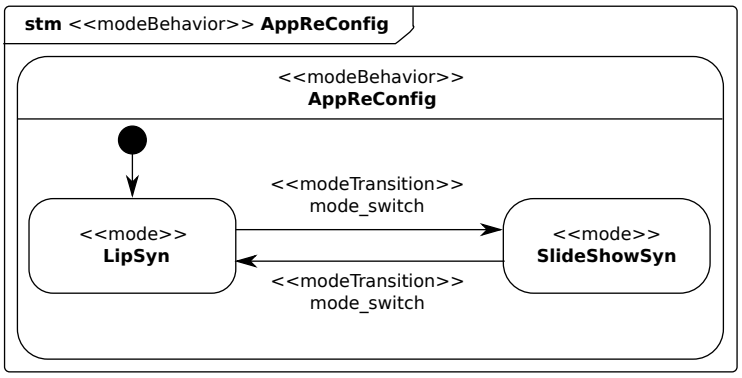


Figure 6.5: The FSM for application reconfiguration.

6.6). Processor p_1 has two operating frequencies 15 and 45 MHz, and processor p_2 and hardware accelerator *acc* have frequency values 30 and 45 MHz. The *Hardware Resource Modeling* package of MARTE is used here to describe the architecture. Figure 6.6 gives the details of our modeling.

Reconfiguration modeling. The reconfiguration of the considered platform is induced by the changes of frequency values of the three processing elements. E.g., a possible platform configuration is that p_1, p_2 and *acc* all have the same frequency 45 MHz. Such an configuration can be modeled by using the stereotype $<< configuration >>$, similar to the application configuration in Figure 6.4. The reconfiguration behavior is under the control of the manager, and can be modeled by an FSM as well. Since different application configurations usually require different platform configurations to meet system requirements, the platform reconfiguration depends on the application reconfiguration. Step 2 will describe how to choose the platform configuration for each application configuration, while its reconfiguration behavior will be modeled in Step 3.

Step 2: Configuration Design

For each application configuration, the designer needs to choose a platform configuration and compute a mapping configuration so that the implementation meets the system functional and non-functional requirements. The CLASSY framework in Chapter 4 has been proposed to address this design issue. We thus need to transform the MARTE models of application configurations and platform configurations into the clock models that CLASSY takes as input. W.r.t. each application configuration, the CLASSY design space exploration then can be performed to compute a set of Pareto-optimal mapping solutions.

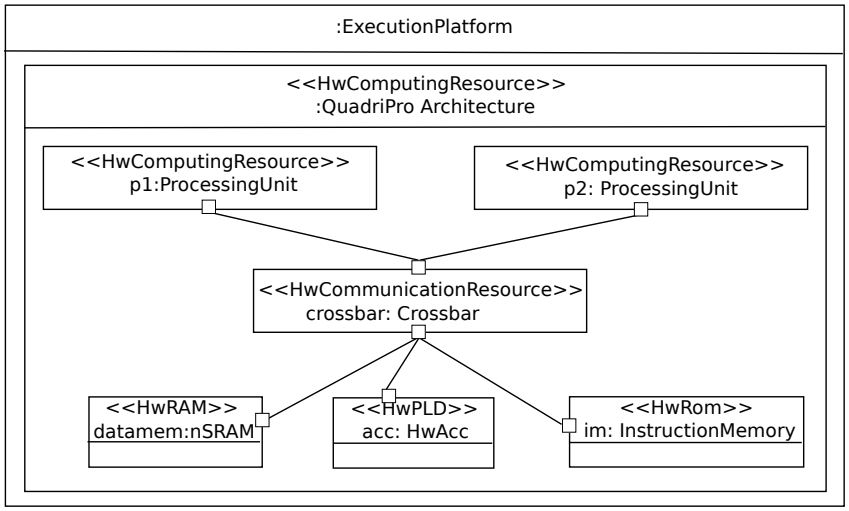


Figure 6.6: The hardware platform model.

Transformation of the MARTE models into abstract clocks.

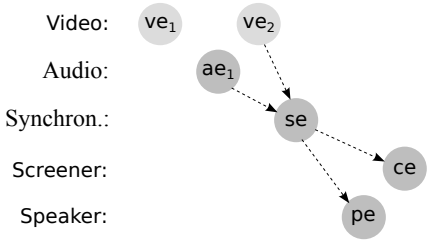


Figure 6.7: The abstract clock modeling of the lip synchronization mode.

Application configuration. For the two application configurations, i.e., lip and slide show synchronizations, the processing and display of a playable unit form an iteration. Figure 6.7 shows the abstract clock modeling of one iteration of the lip synchronization. The slide show synchronization can be modeled similarly.

Platform behavior. We take the platform configuration $p_1(15), p_2(30), acc(45)$ as an example. Figure 6.8 depicts the clock model of the configuration w.r.t. a reference or ideal clock, which is defined by computing the Least Common Multiple (LCM) in order to synchronize multiple clocks.

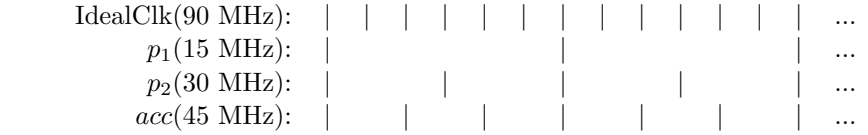


Figure 6.8: Clock modeling of a hardware platform configuration w.r.t an ideal clock. The values in the brackets denote the frequency of the resources.

ID	Mapping and Platform Solutions (task/PE(the used frequency))	Time (ms)	Energy (joules)
1	video/acc(45) audio/acc(45) synchron/acc(45) screener/p2(30) speaker/p1(45)	200.0	40.0
2	video/p1(45) audio/p2(45) synchron/p1(45) screener/p2(45) speaker/p2(45)	444.4	12.0
3	video/p1(45) audio/p2(45) synchron/p2(45) screener/p2(45) speaker/p1(45)	400.0	13.0
4	video/p1(45) audio/p2(45) synchron/p2(45) screener/p2(45) speaker/p2(45)	511.1	11.0
5	video/p1(45) audio/acc(30) synchron/p2(45) screener/p2(45) speaker/p2(45)	377.8	19.0
6	video/p1(45) audio/acc(30) synchron/acc(30) screener/p2(45) speaker/p2(45)	266.7	22.0
7	video/acc(30) audio/acc(30) synchron/p2(45) screener/p1(45) speaker/p2(45)	244.4	37.0

Table 6.1: The Pareto-Optimal solutions for the LipSync configuration computed by CLASSY. The time and energy costs are the values per iteration.

Design analysis. To perform the design analysis, elementary costs, e.g., the time and energy costs of each event on each PE, must be defined. CLASSY is then used for design space exploration for each application configuration, and produces a number of design solutions with a trade-off of resource use, time and energy costs. Each generated design solution of CLASSY consists of a mapping and a platform configuration, i.e., how the tasks are mapped onto the platform resources, and what frequency values are used for the computing resources.

We also take the LipSync configuration as an example. By using CLASSY (random values are used for the elementary costs), we got 7 Pareto-optimal solutions as shown in Table 6.1. Among the computed mapping solutions, we take three mapping configurations: configuration 1 with the best performance (denoted by $LipBestPerf$), configuration 4 with the least energy consumption (denoted by $LipBestEnj$), and configuration 5 with a good trade-off (denoted by $LipTradeOff$) for the implementation solutions of the LipSync configuration. A run-time manager, designed in the next section, would decide on which choice to use depending on the run-time situations.

Similarly, the design analysis can be performed for the SlideShowSync configuration. We also suppose that three configurations, namely $SlideBestPerf$, $SlideBestEnj$, and $SlideTradeOff$ are chosen as implementation candidates.

Step 3: Reconfiguration Manager Design

Modeling of mapping and platform reconfiguration. For each application configuration, we have computed three mapping and platform configurations or implementations. The choice among the implementations for each application configuration should be decided at run-time by the manager according to run-time situations. The implementation reconfiguration behavior thus needs to be modeled, so that it can be taken into account in the manager design process.

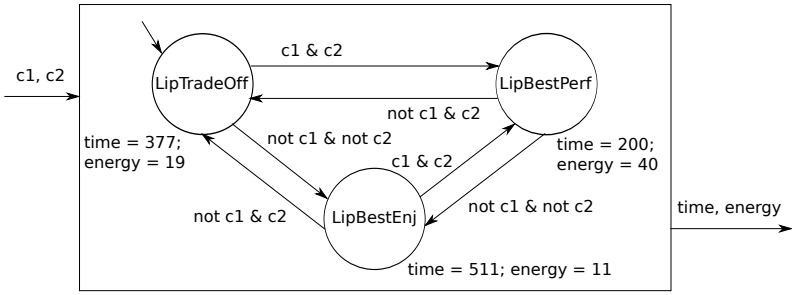


Figure 6.9: The implementation reconfiguration automaton for the LipSync mode.

We take the LipSync configuration and its three computed implementation candidates as an example. Figure 6.9 shows the reconfiguration automaton model. The transitions between the implementations are considered to be controllable, denoted by controllable variables $c1$ and $c2$. Each state represents a corresponding mapping and platform configuration. The characteristics of each implementation, i.e., execution time and energy consumption, are also associated with each state. The guards in terms of the controllable variables of the transitions are encoded in such a way that the best performance mode is preferred when it has more than one valid transition choices. This is because that the generated controller by the BZR compiler always evaluates true to the controllable variables if possible (as described in Section 3.4 of Chapter 3).

BZR encoding of the system reconfiguration behavior and DCS. The composition of the reconfiguration models of application, mapping and platform behaviors comprises the system reconfiguration behavior. The application reconfiguration behavior modeled by a FSM in Step 1 can be transformed into an automaton easily. All the automata models are then encoded by the BZR programs. Regarding controllability of reconfiguration behaviors, the application reconfiguration is uncontrollable, but depends on user commands, while the implementation (i.e., mapping and platform) reconfiguration is controllable. In the case study, the system has two boolean uncontrollable inputs: mode switch denoted by *stc*, and high battery level denoted by *high*. The manager to be designed needs to decide on the implementation solutions and control their reconfigurations w.r.t. the application configurations and system requirements by using the defined controllable variables, i.e., $c1$ and $c2$ of Figure 6.9.

As mentioned in the informal description in Section 6.2.2, there are two system requirements:

1. the energy consumption is constrained by the battery level;
2. among the implementation configurations that meet requirement 1), the one with the best performance should be used.

```
contract
var control_rule:bool;
let
 control_rule = if high then energy <= 40
                else energy <= 20;
tel
assume true
enforce control_rule
with (c1,c2:bool)
```

Figure 6.10: The BZR contract enforcing system requirements.

Figure 6.10 gives a contract describing an control objective corresponding to requirement 1). In consideration of the implementation configurations in Figure 6.9, when the energy level is *high*, all the three configurations are valid as the energy constraint is 40. The $LipBestPerf$ implementation becomes invalid when the energy level is not *high*. Requirement 2) is captured by the way of encoding the controllable variables in Figure 6.9 as discussed above.

At last, by feeding the BZR program to the BZR compiler, it synthesizes a manager automatically satisfying the two system requirements. Figure 6.11 depicts a simulation scenario by using the graphical display tool sim2chro[1]. It simulates the system behavior by combining the generated manager with its original BZR models.

Initially, at instant 1, the system is in the *LipSync* mode, denoted by variable *lip* equal to true in the figure, and the energy level is not high, denoted by *high* equal to false. According to the contract, the upper bound of energy cost is currently 20. Configurations LipTradeOff and LipBestEnj are thus valid. Considering the requirement 2), the trade-off configuration with time cost 377 and energy cost 19 is chosen by the manager. At instant 4, the energy level becomes *high*, which implies the upper bound of energy cost becomes 40. According to the two requirements, the $LipBestPerf$ configuration should be chosen by the manager. In the figure, it only takes effect one instant later, i.e., at instant 5, since the weak transition "until" is used in the BZR encoding (as described in Section 3.4 of Chapter

[1] http://www-verimag.imag.fr/~raymond/edu/distrib/

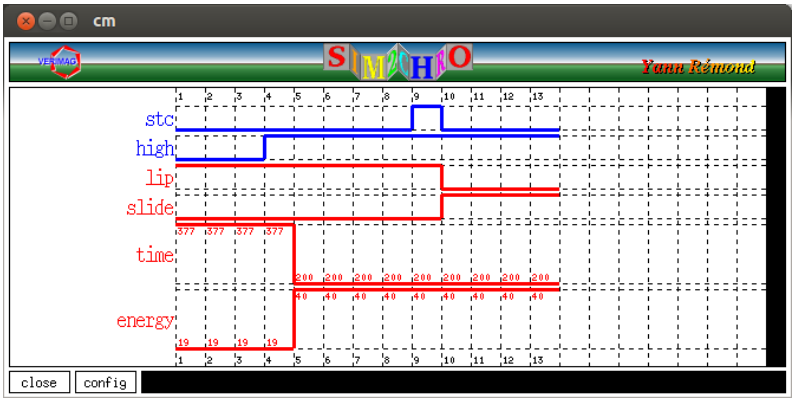


Figure 6.11: A simulation scenario of the controlled system.

3). This is also the case at instant 9. The switch *stc* becomes true, and the system goes to the *SlideSync* configuration, denoted by *slide* becomes true and *lip* becomes false, at instant 10.

Step 4: Analysis Result Integration

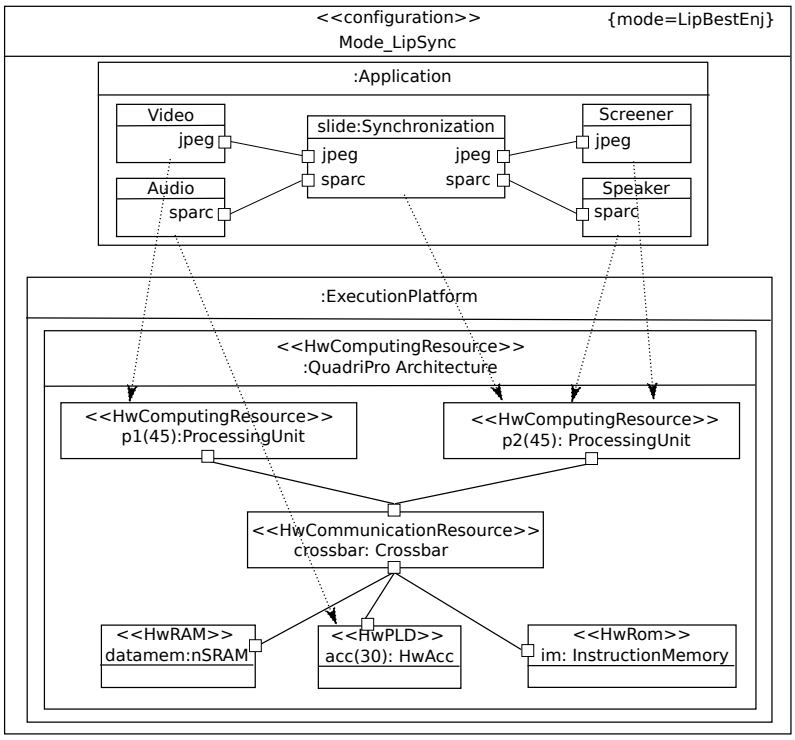


Figure 6.12: The *LipBestEnj* implementation model of the *LipSync* configuration.

In Steps 2 and 3, the implementation candidates (e.g., Table 6.1) and their reconfiguration behavior model (e.g., Figure 6.9) for each application configuration are computed. These results should be modeled and integrated into the original MARTE models so that

existing tools like Gaspard2 [Gamatié *et al.* 2011] can be used for generating low level codes such as synchronous language Signal, VHDL for further analysis. Take the *LipBestEnj* implementation of the *LipSync* configuration as an example. It can be modeled by using the stereotype $<< configuration >>$ as shown in Figure 6.12. The automaton models for the implementation reconfiguration, e.g., Figure 6.9, can also be transformed into FSM models.

All the MARTE models for system application and implementation (i.e., platform and mapping) behavior comprise the uncontrolled system behavior. They can feed tools like Gaspard2 for generating low level codes for further analysis. To have the final controlled system, the manager generated by BZR should also be integrated into the system implementation. BZR can generate the manager in C or Java directly. The manager can be integrated with the generated low level code of Gaspard2 from the MARTE models for combined analysis and/or implementation. The details on how to integrate the generated manager with the system have been described in Section 5.6 of Chapter 5.

6.3 CLASSY for the Design and Evaluation of Run-Time Managers

Adaptive systems require run-time managers to control their adaptive behaviors. Therefore, there exists a need to develop a simulation framework to explore and evaluate run-time managers for adaptive systems. CLASSY can serve as such a modular simulation framework allowing the analysis and evaluation of various designed managers for adaptive MPSoCs. In this section, we show how to use CLASSY as a simulation and analysis framework for designed managers by using an illustrative example. DCS used in Chapter 5 will be employed to design a run-time manager.

Section 6.3.1 describes the illustrative example informally. Section 6.3.2 applies DCS to design a run-time manager. Section 6.3.3 integrates the designed controller into the simulation process of CLASSY for analysis and evaluation.

6.3.1 An Illustrative Example

We consider a data-flow *application* whose functionality is described by the combination of a task graph and automata, as shown in Figure 6.13. Tasks B and C in the graph have two different execution modes or configurations, described by automata. The execution mode of task B is data-dependent, i.e., depends on the run-time incoming data type. The two modes of task C are designed to have different energy costs: mode $c1$ consumes less than $c2$. A manager needs to decide for task C which mode to use at run-time.

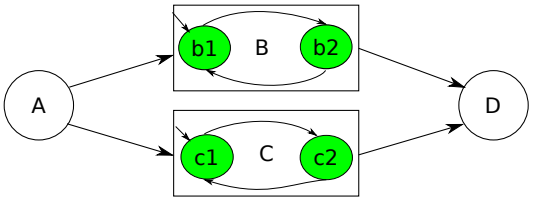


Figure 6.13: System functionality described by combining a task graph and automata.

An MPSoC *platform* consisting of two PEs p_1, p_2 is considered to implement the application. Both PEs have two operating frequencies and can switch between them dynamically. p_1 can operate on frequency $20MHz$ or $30MHz$, while p_2 operates on $45MHz$ or $60MHz$.

Furthermore, a battery is considered to power the platform, and two battery levels: high and low are distinguished.

Regarding *mapping*, it is assumed that task A can only be mapped onto p_1, and D on p_2, while tasks B and C can be mapped onto both processors and are able to migrate between the two processors.

We consider the following *adaptation policy* that concerns three adaptive aspects of the system:

1. application aspect: mode $b2$ of task B and mode $c1$ of task C cannot be active in the same iteration; task C should stay in mode $c1$ that consumes less energy if possible.

2. platform aspect: the two PEs cannot use high frequency at the same time when the battery level is low; the user always expects the best performance, i.e., two PEs use their high frequency values if possible.

3. mapping aspect: tasks B and C cannot be mapped onto the same PE for efficiency; when C is in c2 mode, it must be mapped onto a PE running at its high frequency value; tasks B and C should not migrate to avoid migration costs if possible.

6.3.2 Run-Time Manager Design

We employ the design technique of Chapter 5 to address the manager design of the illustrative example. Firstly, we model its reconfiguration behavior in terms of parallel automata. The existing BZR tool is then used to perform the DCS and generate the manager automatically.

Modeling Reconfiguration Behavior by Automata

Application Reconfiguration. The application reconfiguration comes from the two reconfigurable tasks B and C. The reconfiguration of task B depends on the input data, which is uncontrollable. The reconfiguration of task C is decided by the manager, and thus is controllable. Figure 6.14 shows their automata models. For task B, when dt is true, which represents that the incoming data is of some type, mode $b2$ is used. Otherwise, mode $b1$ is used. For task C, a controllable variable c_md is used to guard the transition between its two modes.

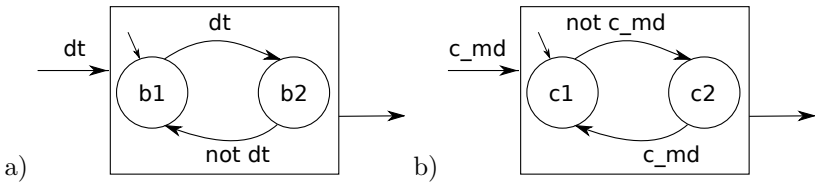


Figure 6.14: Reconfiguration models a) and b) for tasks B and C, respectively.

Platform Reconfiguration. The two PEs p_1 and p_2 can switch between their two frequencies, which is under the control of the manager. Figure 6.15 shows their automata models. Controllable variables c_up1 and c_up2 are used to guard the switches. States hf_pi and $lf\_pi, i = 1, 2$ represent that the corresponding PE p_i is using respectively the high frequency and low frequency value.

Regarding the battery, we suppose a sensor is used to detect its battery state by emitting *down* and *up* signals. We thus model the battery behavior by Figure 6.16.

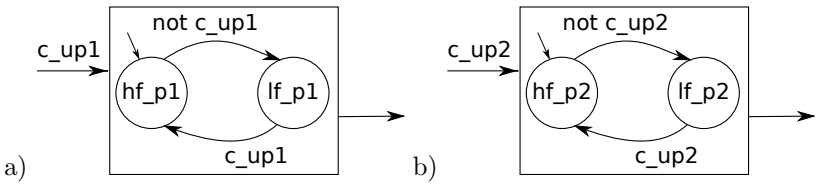


Figure 6.15: Reconfiguration models of PE p_1 and p_2.

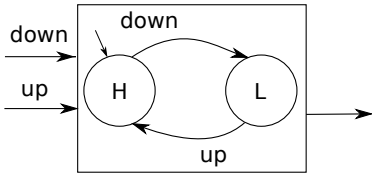


Figure 6.16: The battery model.

Mapping Reconfiguration. The mapping of tasks A and D is fixed, and thus does not introduce reconfiguration. Tasks B and C can be mapped onto the two PEs, and can migrate from one to the other. The migration decision, i.e., whether to migrate, is decided by the manager at run-time. Figure 6.17 gives the mapping behavior models of tasks B and C. Take Figure 6.17 a), the mapping behavior of task B, as an example. Initially, B is not mapped, i.e., in state idle denoted by I. Once it is requested, denoted by rb, the controller decides which PE to run it, denoted by states $B/p1$ and $B/p2$, by using controllable variable c. At the next reconfiguration point, e.g., after the execution of some iterations, when B is in state $B/p1$ or $B/p2$ and requested again (denoted by rb), the manager should decide whether to migrate it by using controllable variable c. In the model, B migrates if c is false. The mapping behavior of task C is modeled similarly as in Figure 6.17 b).

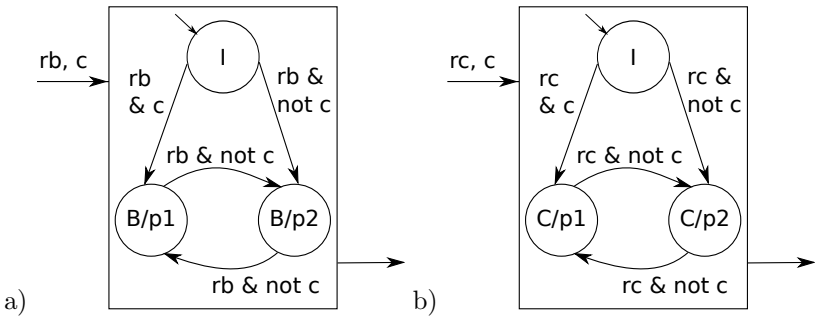


Figure 6.17: Mapping models of tasks B and C.

Modeling of Adaptation Policy. The adaptation policy then needs to be modeled in terms of the states and transitions of the above defined autamata. In the automata models of this section, we did not show their Boolean output variables representing their states for simplicity. In the following, we use the notations labeled on the states as the Boolean output variables to represent whether the states are active. For instance, in Figure 6.14 a), Boolean variable $b1$ represents that state b1 is active if variable $b1$ is true.

1. Application aspect:

   - states b2 and c1 cannot be active in the same iteration: $\neg(b2 \bigwedge c1)$.

- task C should stay in state c1 if possible: this is captured by the way of encoding the transition guard with controllable variable *c_md* of Figure 6.14 b). This is because that the manager computed by BZR would evaluate the controllable variable *c_md* to true, when both true and false values are allowed for *c_md* (as described in Section 3.4 of Chapter 3). In this way, when task C is in state c1, it would stay in state c1 if both states c1 and c2 are allowed, while when it is in state $c2$, it would go to state c1 even if staying state c2 is also allowed.

2. Platform aspect:

   - the two PEs cannot use their high frequency values at the same time when the battery level is low: $\neg(hf\_p1 \bigwedge hf\_p2 \bigwedge L)$.
   - the user always expects the best performance, i.e., two PEs use high frequency values if possible: similar to the reconfiguration behavior model of task C in Figure 6.14 b), this is captured by the way of encoding the guards with the controllable variables *c_up*1 and *c_up*2 of Figure 6.15.

3. Mapping aspect:

   - tasks B and C cannot be mapped onto the same PE for efficiency: $\neg(B/p_1 \bigwedge C/p_1) \bigwedge \neg(B/p_2 \bigwedge C/p_2)$.
   - when C is in mode/state c2, it must be mapped onto a PE running at its high frequency value: $(c2 \bigwedge C/p_1 \bigwedge hf\_p1) \bigvee (c2 \bigwedge C/p_2 \bigwedge hf\_p2)$.
   - tasks B and C should not migrate to avoid migration costs if possible: this is captured by the transition guards with controllable variable c between the two mapped states $B/p1$ and $B/p2$ (respectively states $C/p1$ and $C/p2$) of Figure 6.17, similar to Figure 6.14 b). The generated manager by BZR evaluates true to variable c if both true and false values are allowed.

BZR Encoding and Controller Generation

Given the automata models of Section 6.3.2, they can be easily programmed in BZR, as shown in Section 3.4 of Chapter 3. We do not give the code here. The BZR compiler is then able to generate a manager automatically. As CLASSY is developed in Java, we choose Java as the target code for the generated manager. In the next section, the manager will be combined with CLASSY for simulation and analysis.

6.3.3 Simulation and Analysis of the Designed Manager by using CLASSY

This section shows how a designed manager can be integrated into the CLASSY framework for simulation and analysis. We firstly adapt the scheduling algorithm proposed in Chapter 4 to better address the scheduling of adaptive embedded systems. Then, we use the illustrative example and the generated manager by BZR to illustrative how it works.

As presented in Algorithm 1 in Section 4.4.3 of Chapter 4, CLASSY schedules an (adaptive) system in an iteration-based way. It interacts with the reconfiguration manager and applies possible reconfiguration decisions before the scheduling of each iteration. Moreover, Algorithm 1 distinguishes an application controller $\mathcal{C}(B_T)$ and a platform controller $\mathcal{C}(B_P)$ dedicated to the application and platform reconfiguration management respectively. It does not take into account mapping reconfiguration or task migration behavior. In the following, we improve Algorithm 1 by

- integrating a single run-time manager which manages the reconfiguration behaviors concerning all the three aspects (i.e., application, platform and mapping) of the considered adaptive system, and
- allowing the user to define a reconfiguration checking interval in terms of the number of iterations, which represents the frequency that the scheduler interacts with the reconfiguration manager and applies reconfiguration actions if demanded.

Algorithm 4 shows the principle of the improved scheduling algorithm. An input variable $reconfigInterval$, allowing the user to define the reconfiguration checking interval, and a local variable $scheduledIteratinNum$, representing the number of iterations that has been scheduled, are defined. ω represents the number of iterations to be scheduled, as defined in Algorithm 1. $\mathcal{M}$ represents the run-time manager responsible for managing the system reconfiguration behavior.

Algorithm 4 An Improved Scheduler

```
1:  scheduledIterationNum = 0;
2:  while true do
3:    M.collectInformation();
4:    M.computeNewConfiguration();
5:    if reconfiguration happens then
6:      add reconfiguration costs accordingly;
7:    end if
8:    if scheduledIterationNum+reconfigInterval>ω then
9:      schedule (ω-scheduledIterationNum) iterations based on the current configuration;
10:     break;
11:   else
12:     schedule reconfigInterval iterations based on the current configuration;
13:   end if
14:   scheduledIterationNum = scheduledIterationNum + reconfigInterval;
15: end while
```

At each reconfiguration checking point, the manager $\mathcal{M}$ firstly collects relevant information from the run-time situations, denoted by function $collectInformation()$, and computes a new configuration accordingly, denoted by $computeNewConfiguration()$, as shown in Lines 3 and 4. Line 5 checks if some reconfiguration happens, i.e., the new computed configuration is different from the previous one. If it does, the corresponding time and energy costs of the reconfiguration actions would be added accordingly. From Line 8 to Line 13, a number of iterations is scheduled based on the current system configuration as does in Algorithm 1. Line 8 checks if the remaining number of iterations is less than $reconfigInterval$. If it is, the rest iterations would be scheduled and the scheduler finishes its scheduling. Otherwise, $reconfigInterval$ iterations are scheduled, and the scheduler reaches the next reconfiguration checking point. Line 14 updates the scheduled number of iterations.

Next, we use this algorithm to simulate the illustrative example, whose reconfiguration is controlled by the manager generated by BZR. In the example system, the factors that affect its reconfiguration behavior are the input data type dt, the battery level indicating signals up and $down$, and the requests rb and rc of tasks B and C. The BZR generated manager in Section 6.3.2 has a step function, which takes the inputs $dt, down, up, rb, rc$ and computes the outputs indicating the new configuration. The outputs are: boolean variables $b1, c1$ for indicating the modes of tasks B and C, hf_p1, hf_p2 for indicating the frequency

values of PEs p_1 and p_2, and $tBonP1, tBonP2, tConP1, tConP2$ for indicating the mapping configurations of tasks B and C. To integrate the BZR manager, we replace the Line 3 and Line 4 of Algorithm 4 with the step function of the manager. The outputs are then interpreted to set the new configuration for the following scheduling.

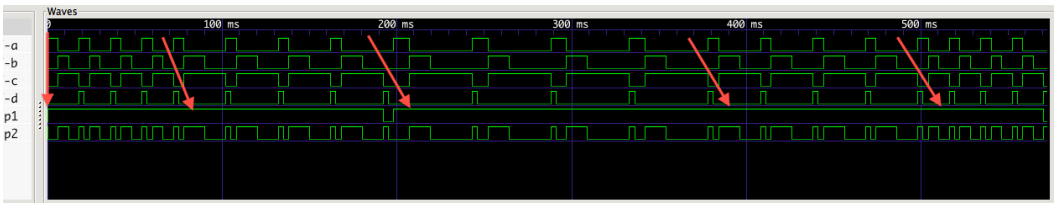


Figure 6.18: The first simulation scenario of CLASSY combined with BZR generated controller.

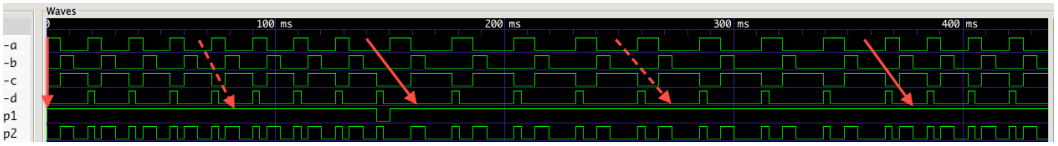


Figure 6.19: The second simulation scenario of CLASSY combined with BZR generated controller.

Figures 6.18 and 6.19 give two CLASSY simulation scenarios shown by GTKWave, with 20 iterations are scheduled and 4 iterations as the reconfiguration checking interval. Waves $-a, -b, -c, -d$ represent respectively the executions of tasks A, B, C and D on their corresponding mapped PEs, while waves p_1, p_2 represents respectively the usages of PEs p_1 and p_2, with up means being executing, and down means being idle.

PEs	A	b1	b2	c1	c2	D
p_1	1	2	4	2	4	-
p_2	-	2	4	2	4	1

Table 6.2: The input time costs in terms of PE cycles of tasks on PEs.

Table 6.2 gives the input time costs of the tasks on the PEs. The costs of task A on p_2 and task D on p_1 are not given, as they are not allowed to be mapped on the corresponding PEs. The communication costs and reconfiguration costs are all set to 0. Random functions are used to evaluate values to the inputs dt, $down$ and up. The requests rb and rc are set to true at each reconfiguration checking point.

The arrows and dotted arrows in Figures 6.18 and 6.19 indicate, during the executions of the four tasks, the checking points where the scheduler interacts with the manager. The arrows indicate the points that the system is demanded to reconfigure, while dotted arrows indicate the points that the system is demanded to keep the current configuration. Both scenarios have five reconfiguration checking points, say reconfiguration 0 to 4. Table 6.3 shows the inputs and computed configurations of the manager at each reconfiguration checking point of Figure 6.18. The manager computes the next system configuration based on the inputs and the current configuration. For instance, at reconfiguration point 1, the system configuration is computed based on the inputs at reconfiguration point 1 and the system configuration at reconfiguration point 0.

Reconfig.	inputs (*dt*, *down*, *up*)	system configurations
0	(false, false, true)	(A/p1(30), b1/p2(60), c1/p1(30), D/p2(60))
1	(true, false, true)	(A/p1(30), b2/p2(60), c2/p1(30), D/p2(60))
2	(true, true, false)	(A/p1(20), b2/p2(60), c2/p1(20), D/p2(60))
3	(true, false, true)	(A/p1(30), b2/p2(60), c2/p1(30), D/p2(60))
4	(false, false, true)	(A/p1(30), b1/p2(60), c1/p1(30), D/p2(60))

Table 6.3: The inputs and computed system configurations at each reconfiguration checking point.

6.4 Summary and Discussion

This chapter presented two ways of combining the proposed design frameworks in Chapters 4 and 5 for adaptive MPSoCs. Firstly, it combined the two design frameworks to construct a design flow for adaptive MPSoCs. The design flow starts from the MARTE high level system modeling, and then employs the two proposed design frameworks to respectively tackle the two design issues, i.e., the design of each system configuration and the derivation of a reconfiguration controller. At last, the design results are integrated into the original modeling framework, and existing tools such as Gaspard2 can be used to generate low level codes for further analysis and implementation.

There exist many high level approaches to deal with the two design issues of adaptive MPSoCs. Some approaches focus on a single design aspect, e.g., [Stuijk *et al.* 2010] [Abdallah *et al.* 2012] for the design of a configuration or scenario, and [Smit *et al.* 2005] [Ghaffari *et al.* 2007] for the run-time reconfiguration management. These approaches need to be combined with each other to form complete design flows for adaptive MPSoCs. Other approaches also address the two design issues of adaptive MPSoCs together, e.g., [Schor *et al.* 2012] [Schranzhofer *et al.* 2010] [Singh *et al.* 2011]. However, they usually focus on the management of one or two adaptive aspects i.e., mapping and/or platform reconfiguration management, and consider manual encoding of the run-time managers. Our combined approach advocates correct-by-construction, and can deal with the management of all the three adaptive aspects. The combined design flow bears the same limitations of each design framework, e.g., the ability to capture cyclic task graphs for the design of one configuration, and the scalability for the manager derivation. In addition, to combine the two design frameworks, manual encoding is required to connect the results of one framework to the other, which is tedious, and requires the users to know both frameworks well.

Secondly, it presented how the CLASSY framework of Chapter 4 could serve as a high level simulator and assist the designers to design and evaluate run-time managers. The discrete control technique has been considered to design such a manager, which is integrated into the CLASSY simulation process for analysis and evaluation.

Most frameworks that deal with he design of adaptive systems e.g., [Erbas *et al.* 2007] [Balarin *et al.* 2003] [Stuijk *et al.* 2011] do not (explicitly) separate the specification and design of the reconfiguration manager from that of the uncontrolled system behavior. The work that also employs its design framework as a simulator for analyzing and evaluating different run-time managers is proposed in [Sigdel *et al.* 2012]. However, it focuses on the management of the run-time mapping reconfiguration, and does not take into account the management issue of application and platform reconfigurations. Although we have used DCS to design a manager to perform integrated simulation in CLASSY, any customized manager can be integrated in the framework. In addition to the limitations of the CLASSY framework, the designers currently have to encode customized managers in Java and integrate them in CLASSY manually.

The two combination means presented in this chapter provide interesting supports for different design aspects of adaptive MPSoCs. However, manual encoding has been used to connect the results of one framework to the other. A dedicated tool allowing the extractions of useful information from the results of one framework to the inputs of the other framework would be favorable. E.g., a tool can take the Pareto-optimal mapping results of the CLASSY framework, and build the corresponding reconfiguration automata taking into account the results for the framework of Chapter 5 to perform the reconfiguration manager design.

Chapter 7

Conclusion and Perspectives

Contents

7.1 Conclusion 113
7.2 Perspectives 116

7.1 Conclusion

The work presented in the thesis deals with the safe design of adaptive MPSoCs. Their design is becoming increasingly complex and challenging, because of more and more functionality integrated into embedded systems, the employment of the MPSoC platforms, their adaptivity feature, etc. At the same time, the designers also face strict time-to-market pressures. To manage the increasing design complexity and time-to-market pressures, raising the levels of abstraction and using abstract models for early design analysis, evaluation and validation are becoming a common practice. We advocate cost-effective and safe design methodologies based on high level formal models for adaptive MPSoCs. We believe that such approaches can assist the designers to make the design of adaptive MPSoCs more efficient, and at the same time, bring confidence in the reliability of their design.

We have identified two main design issues for the safe design of an adaptive MPSoC as follows.

- Firstly, design correctness must be addressed to ensure system reliability in every possible system configuration, referred as *the design of a configuration of an adaptive MPSoC* issue. This requires that the proper resource allocation and mapping (i.e., the binding and scheduling of application tasks on the allocated resources) decisions must be made for each system application scenario or configuration w.r.t. system functional and non-functional requirements.

- Secondly, reconfiguration correctness must also be established to safely control the variation between system configurations, referred as *the reconfiguration management* issue. This requires a run-time manager that controls and coordinates the system reconfiguration behavior in reaction to system run-time situations according to system requirements.

The thesis firstly dealt with these two design issues separately in Chapter 4 and Chapter 5 respectively.

- Chapter 4 proposed a high-level framework named CLASSY for the rapid and cost-effective design of (adaptive) applications on MPSoCs. A multi-clock modeling of both software and hardware has been considered by exploiting the notion of abstract clocks borrowed from synchronous data-flow languages. Our approach enriches the vision of the application of the synchronous model [Benveniste *et al.* 2003] by encoding

the quantitative time via abstract clocks. The resulting model provides a uniform support for design assessment w.r.t. quantitative properties. Our approach is an ideal complement to lower-level design assessment techniques for MPSoCs, such as physical prototyping and simulation. It also aims to serve as an intermediate reasoning support that is usable, from very high-level MPSoC models (e.g., in UML MARTE profile), to deal with critical design decisions. The framework can be used to deal with the first design issue, i.e., the design of a configuration of adaptive MPSoCs. Furthermore, it is also flexible enough to capture the adaptive behaviors of MPSoCs, and can be used as a high level simulator for adaptive MPSoCs to evaluate customized run-time managers.

The static analysis of data-flow application designs with predictable behaviors, has mainly based on data-flow models, such as Kahn Process Networks (KPNs) and Synchronous Data-Flows (SDFs). KPNs are expressive enough to capture dynamic application behaviors, but the expressiveness power also makes it difficult to predict their precise behaviors over time. Existing techniques based on KPNs, such as [Sigdel 2011], [Erbas *et al.* 2007], [Schor *et al.* 2012] usually employ simple run-time scheduling strategies e.g., first come first served algorithms, and do not investigate the design of scheduling algorithms. In our framework, we have studied the requirements for admissible scheduling requirements, and proposed a correct by construction scheduling algorithm. The relative simple and static nature of SDFs makes it possible to develop design-time analysis techniques (e.g., [Stuijk *et al.* 2008] [Ghamarian *et al.* 2006]), as well as efficient scheduling strategies (e.g.,[Stuijk 2007]) [Stuijk *et al.* 2011]. However, SDFs are not expressive enough to capture dynamic application behaviors. Abstracting away the system dynamic behavior by using SDFs would overestimate resource requirements [Stuijk *et al.* 2010]. The Scenario-Aware Dataflow (SADF) MoC has thus been employed (e.g., [Stuijk *et al.* 2010], [Stuijk *et al.* 2011] to combine a finite state machine-like structure with SDFs to deal with the modeling and analysis of adaptive applications. Compared to these SDF and SADF based approaches, they do not investigate the impact of potential delay between processor cycles on scheduling. The major limitation of our approach is the supported application models: applications described as cyclic component graphs are not currently addressed. In addition, the storage distribution computed by our design space exploration framework for each application mapping solution is not optimal. Third, the manual transformation from MARTE described application and platform specifications to our reasoning framework is tedious, and can become very complex and error-prone if the problem becomes big. An automatic transformation has not been developed.

- Chapter 5 presented a general framework, based on a tool-supported synchronous variant [Marchand & Samaan 2000] of the *discrete control* [Ramadge & Wonham 1989], for the autonomic management of adaptive MPSoCs. It favors automatic and correct-by-construction manager derivation. We have focused on adaptive MPSoCs implemented on reconfigurable architectures especially DPR FPGAs, which can draw various benefits such as efficiency and flexibility. Such architectures constitute a platform for adaptive computing that is gaining widespread use. In the framework, the system reconfiguration behavior is modeled in terms of synchronous parallel automata. The reconfiguration management computation problem w.r.t. multiple objectives regarding e.g., resource usages, performance and power consumption is encoded as a discrete controller synthesis (DCS) problem. The existing BZR programming language and Sigali tool are employed to perform DCS and generate a controller that satisfies the

system requirements. We have performed extensive experiments to evaluate the scalability of our approaches, and presented an experimental validation of our proposal by implementing a real-life video processing system on a Xilinx FPGA platform.

Compared to other existing techniques for self-management of adaptive systems, e.g., heuristics and machine learning techniques, the main advantage of control techniques is that they are able to provide formal correctness and/or performance guarantees. The authors in [Maggio *et al.* 2012] discuss some existing approaches applying standard control techniques such as Proportional Integral and Derivative controllers or Petri nets. However, discrete control has only been seldom applied [Guillet *et al.* 2012], and to the best of our knowledge, only few works have targeted at computing systems on reconfigurable architectures. One of the first implementations of a FPGA-based self-aware adaptive system is proposed in [Sironi *et al.* 2010]. It adopts the application heartbeats as its monitoring framework, and a heuristic mechanism to switch between configurations. Self-management in the form of self-healing exploiting FPGAs is proposed in [Jovanović *et al.* 2008]. However, the approach does not involve control. Another architectural proposal in [Majer *et al.* 2007] provides a slot-based organisation of the reconfigurable hardware and an elaborate communication framework with good reconfiguration support. The focus is, however, on infrastructure aspects rather than on control. Our approach is closer to that in [Eustache & Diguet 2008], but we focus on logical aspects and discrete control. In [Quadri *et al.* 2010b], a design flow, from high level models to automatic code generation, for the implementation of reconfigurable FPGA based SoCs is proposed. The system control aspects need to be modeled manually and integrated into the flow, while we advocate automatic controller synthesis. Compared to [Guillet *et al.* 2012], we have applied more elaborate DCS algorithms, and the integration into a design flow and compilation chain is more developed. The major concern of our approach is the scalability issue, which is common to other formal techniques like model checking. Besides, our models did not take into account some other interesting aspects, such as communication. Third, the automatic transformation from the MARTE described system specification to the automata models employed in our framework, though seeming direct, has not been developed.

At last, we investigated the possible ways of combining the two proposed design frameworks for adaptive MPSoCs in Chapter 6. The following two ways of combinations have been presented.

- First, they were combined to construct a complete design flow for adaptive MPSoCs. The design flow starts from the MARTE high level system modeling, and then employs the two proposed design frameworks to respectively tackle the two design issues. At last, the design results are integrated into the original MARTE modeling framework, with which existing tools such as Gaspard2 [Gamatié *et al.* 2011] can be used to generate low level codes for further analysis and implementation.

  There exist many high level approaches to deal with the two design issues of adaptive MPSoCs. Some approaches focus on a single design aspect, e.g., [Stuijk *et al.* 2010] [Abdallah *et al.* 2012] for the design of a configuration/scenario, and [Smit *et al.* 2005] [Ghaffari *et al.* 2007] for the run-time reconfiguration management. These approaches need to be combined with each other to form complete design flows for adaptive MPSoCs. Other approaches also address the two design issues of adaptive MPSoCs together, e.g., [Schor *et al.* 2012] [Schranzhofer *et al.* 2010] [Singh *et al.* 2011]. However, they usually focus on the management of one or two adaptive aspects i.e., mapping and/or platform reconfiguration management, and consider manual encoding of

the run-time managers. Our combined approach advocates correct-by-construction, and can deal with the management of all the three adaptive aspects. The combined design flow bears the same limitations of each design framework, e.g., the ability to capture cyclic task graphs for the design of one configuration, and the scalability for the manager derivation. In addition, to combine the two design frameworks, manual encoding is required to connect the results of one framework to the other, which is tedious, and requires the users to know both frameworks well.

- Second, they were combined to present how the CLASSY framework could serve as a high level modular simulator and assist the designers to design and evaluate run-time managers. To do this, the second framework based on the discrete control technique is used for designing a run-time manager, which is integrated into the CLASSY simulation process for analysis and evaluation.

Most frameworks that deal with he design of adaptive systemts e.g., [Erbas *et al.* 2007] [Balarin *et al.* 2003] [Stuijk *et al.* 2011] do not (explicitly) separate the specification and design of the reconfiguration manager from the uncontrolled system behavior. The work proposed in [Sigdel *et al.* 2012] also employs their design framework as a simulator for analyzing and evaluating different run-time managers. However, it focuses on the management of the run-time mapping reconfiguration, and does not take into account the management issue of application and platform reconfigurations. Although we have used DCS to design a manager to perform integrated simulation in CLASSY, any customized manager can be integrated in the framework. In addition to the limitations of the CLASSY framework, the designers currently have to encode customized managers in Java and integrate them in CLASSY manually.

7.2 Perspectives

The two design frameworks and their combinations presented in the thesis offer some interesting supports for the design of adaptive MPSoCs. However, in order to increase their applicability, some issues need to be researched further.

- W.r.t. the CLASSY framework of Chapter 4 for the design of a configuration of an adaptive MPSoC:
  - CLASSY currently does not support the modeling of applications described as cyclic component graphs. One possible solution to deal with this is to introduce a memory operator in our modeling framework for applications.
  - The storage distribution computed by CLASSY for each application mapping solution is not optimal. A research direction is to combine our framework with the SDF^3 toolset [Stuijk *et al.* 2006a] to address this issue.
  - The autonomic transformation from MARTE described application and platform specifications to our reasoning framework has not been developed. A manual transformation is tedious, and can become very complex and error-prone if the problem becomes big. Some inspirations from [Abdallah 2011] will be taken to deal with it.
- W.r.t. the discrete control based design framework of Chapter 5 for the reconfiguration manager design:

  - Its major concern is the scalability issue. An interesting means is to exploit the modular synthesis and compilation [Delaval *et al.* 2010], which allows the users to decompose big problems in a way that breaks down the combinatorial complexity. It is also interesting for specification and model structuring purposes.
  - The automatic transformation from the MARTE described system specification to the automata in our framework has not been developed. Inspirations from [Guillet 2012] will be taken to deal with this.
  - Another perspective is to enrich our models by taking into account other realistic aspects, such as communication and memory access costs.
  - We are also interested in applying our framework to the control of other real-life embedded systems, such as some video monitoring systems, which monitor the presences and movements of objects and perform corresponding reconfigurations.

- W.r.t. the combination of the two design frameworks: the two combinations have involved manual encoding to connect the results of one framework to the other. A dedicated tool allowing the extractions of useful information from the results of one framework to the inputs of the other framework would be favorable. E.g., a tool can take the Pareto-optimal mapping results of the CLASSY framework, and build the corresponding reconfiguration automata taking into account the results for the framework of Chapter 5 to perform the reconfiguration manager design.

List of Figures

2.1 MPSoC platform template. 10
2.2 STMicroelectronics Nomadik platform, taken from [Wolf *et al.* 2008]. 11
2.3 A 4×4 Mesh topology NoC [Ali *et al.* 2009]. 13
2.4 The Y chart design model, taken from [Tammemäe & Ellervee]. 14
2.5 Illustrative example of Pareto optimality in objective space (left) and the possible relations of solutions in objective space (right), taken from [Zitzler 1999]. 16
2.6 The Y-chart design scheme. 17
2.7 Simplified logic cell implementation. 20
2.8 FPGA with a microblaze softcore. 22
2.9 IBM's Monitor, Analyze, Plan, Execute, Knowledge (MAPE-K) reference model for autonomic control loops, taken from [Huebscher & McCann 2008]. 23

3.1 Skeleton of a Lustre node. 28
3.2 A system configuration modeled by MARTE. 30
3.3 A system reconfiguration behavior modeled by MARTE. 31
3.4 The graphical description of a delayable task. 32
3.5 The textual description of a delayable task. 33
3.6 The textual description of two delayable tasks put in parallel. 33
3.7 An example using encapsulation to enforce the synchronization of two composed automata. 34
3.8 Principle of discrete controller synthesis 35
3.9 A BZR program with contract. 37

4.1 An initial application configuration. 45
4.2 A nominal application configuration. 45
4.3 Application configuration manager. 45
4.4 A multiprocessor platform model. 46
4.5 Set of two behaviors b_1 and b_2. 48
4.6 A platform dynamic behavior. 49
4.7 Execution encoding with ternary clocks. 50
4.8 Execution encodind with ternary clocks. 51
4.9 Overview of the CLASSY tool. 57
4.10 Application behavior for M-JPEG. 60
4.11 Execution times for M-JPEG decoder on an image: CLASSY *vs* SoCLib cycle-accurate simulations (communication via bus and NoC). 61

5.1 Architecture structure. 71
5.2 DAG application specification. 71
5.3 Configurations and reconfigurations. 72
5.4 Models RM_i for tile Ai, and BM for battery. 74
5.5 Scheduler automaton Sdl capturing application execution behaviours. . . . 74

5.6 Execution behavior model TM_A of task A without considering reconfiguration time. . . . 78
5.7 Execution model of task A taking into account reconfiguration overheads. . 79
5.8 A simulation scenario. . . . 86
5.9 The refined task graph for experiments. . . . 87
5.10 The video processing system case study. . . . 89
5.11 Each image is divided into 9 areas for processing, with those covered by grids called corner areas, and the rest ones called cross areas. . . . 89
5.12 The model of system execution mode behavior, with Boolean input ms capturing $ModeSwitch$, and Boolean output h representing whether it is in mode $high$. . . . 90
5.13 The model for choosing filtering algorithms for processing the four corner areas, where inputs $s \in \{true, false\}$ represents that the system gets started or not, $gr \in \{1, 0\}$ represents that the user switches on or off the corner color switch, and outputs $fc \in \{corR, corB, corG, corI\}$ represents the current state and $w \in \{12, 8, 4, 0\}$ represents the weight associated with the state. . 91
5.14 The model for choosing filtering algorithms for processing the five cross areas, where outputs $fc \in \{croR, croB, croG, croI\}$ represents the current state, and $w \in \{15, 10, 5, 0\}$ represents the weight associated with the state. . . . 91
5.15 Global structure of the implementation . . . 93

6.1 The continuous multimedia (CM) player system . . . 98
6.2 The lip synchronization specification. . . . 98
6.3 The slide show synchronization specification. . . . 98
6.4 An application configuration: LipSync. . . . 99
6.5 The FSM for application reconfiguration. . . . 99
6.6 The hardware platform model. . . . 100
6.7 The abstract clock modeling of the lip synchronization mode. . . . 100
6.8 Clock modeling of a hardware platform configuration w.r.t an ideal clock. The values in the brackets denote the frequency of the resources. . . . 100
6.9 The implementation reconfiguration automaton for the LipSync mode. . . . 102
6.10 The BZR contract enforcing system requirements. . . . 103
6.11 A simulation scenario of the controlled system. . . . 104
6.12 The *LipBestEnj* implementation model of the *LipSync* configuration. . . . 104
6.13 System functionality described by combining a task graph and automata. . 105
6.14 Reconfiguration models a) and b) for tasks B and C, respectively. . . . 106
6.15 Reconfiguration models of PE p_1 and p_2. . . . 107
6.16 The battery model. . . . 107
6.17 Mapping models of tasks B and C. . . . 107
6.18 The first simulation scenario of CLASSY combined with BZR generated controller. . . . 110
6.19 The second simulation scenario of CLASSY combined with BZR generated controller. . . . 110

List of Tables

3.1 Examples of some Lustre temporal operators. 28

4.1 Analyzed mapping configurations for M-JPEG. 59
4.2 Profiling data about M-JPEG tasks as inputs for CLASSY. 60
4.3 The experimental results taken from [Stuijk *et al.* 2008]. The unit of throughput is the number of iterations per time unit. The unit of distribution size is the number of data tokens. The results of the example CSDFG is given in a graph, where the maximum and minimum throughputs are not easier to recognized precisely. $>$ and $<$ means respectively sightly bigger and smaller than. The run-time of the experiment is not given. 63
4.4 The experimental results of our framework. The unit of throughput is the number of iterations per time unit. The unit of distribution size is the number of data tokens. λ = (3.8,99.1,99.1,0.02,3.9,0.26,0.26,2.8,4.4,0.39,1.84,1.84). . 63
4.5 The scalability experiment. The unit of throughput is the number of iterations per time unit. 65

5.1 Profiled task implementation characteristics for the working example. . . . 72
5.2 The time costs for DCS operations corresponding to different target objectives w.r.t. different system models and state space sizes. $n : (m_1, ...m_n)$ denotes a model of n functions, with m_i denoting the number of possible implementations of function Fi. $*$ Objectives 7 and 8 are the same kind of operation (objective 8 is thus omitted here). 88

6.1 The Pareto-Optimal solutions for the LipSync configuration computed by CLASSY. The time and energy costs are the values per iteration. 101
6.2 The input time costs in terms of PE cycles of tasks on PEs. 110
6.3 The inputs and computed system configurations at each reconfiguration checking point. 111

Bibliography

[Abdallah *et al.* 2012] A. Abdallah, A. Gamatié, R. Ben Atitallah and J. Dekeyser. *Abstract Clock-Based Design of a JPEG Encoder*. Embedded Systems Letters, IEEE, vol. 4, no. 2, pages 29–32, 2012. (Cited on pages 67, 111 and 115.)

[Abdallah 2011] Adolf Samir Abdallah. *Conception de SoC à Base d'Horloges Abstraites: Vers l'Exploration d'Architectures en MARTE*. PhD thesis, Université de Lille 1, 2011. (Cited on pages 57, 67 and 116.)

[Abdi *et al.* 2011] S. Abdi, G. Schirner, Yonghyun Hwang, D.D. Gajski and Lochi Yu. *Automatic TLM Generation for Early Validation of Multicore Systems*. Design Test of Computers, IEEE, vol. 28, no. 3, pages 10–19, 2011. (Cited on page 18.)

[Adler *et al.* 2007] Rasmus Adler, Ina Schaefer, Tobias Schuele and Eric Vecchié. *From Model-Based Design to Formal Verification of Adaptive Embedded Systems*. In In Proc. of ICFEM 2007. Springer, 2007. (Cited on page 21.)

[Adriahantenaina *et al.* 2003] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez and C.A. Zeferino. *SPIN: a scalable, packet switched, on-chip micro-network*. In Design, Automation and Test in Europe Conference and Exhibition, 2003, pages 70–73 suppl., 2003. (Cited on page 12.)

[Ali *et al.* 2009] Mohammad Ali, Jabraeil Jamali and Ahmad Khademzadeh. *MinRoot and CMesh: Interconnection Architectures for Network-on-Chip Systems*. vol. 54, 2009. (Cited on pages 12, 13 and 118.)

[Altisen *et al.* 2003] Karine Altisen, Aurélie Clodic, Florence Maraninchi and Éric Rutten. *Using Controller-Synthesis Techniques to Build Property-Enforcing Layers*. In Proceedings of the European Symposium on Programming (ESOP'03), pages 174–188, 2003. (Cited on pages 33, 34 and 38.)

[An *et al.* 2011] Xin An, Abdoulaye Gamatié and Éric Rutten. *Safe design of dynamically reconfigurable embedded systems*. In 2nd Workshop on Model Based Engineering for Embedded Systems Design, M-BED'11, Grenoble, France, 2011. (Cited on page 96.)

[An *et al.* 2012a] Xin An, Sarra Boumedien, Abdoulaye Gamatié and Eric Rutten. *CLASSY: a clock analysis system for rapid prototyping of embedded applications on MPSoCs*. In Proceedings of the 15th International Workshop on Software and Compilers for Embedded Systems, SCOPES'12, pages 3–12, 2012. (Cited on pages 4, 45 and 67.)

[An *et al.* 2012b] Xin An, Sarra Boumedien, Abdoulaye Gamatié and Eric Rutten. *CLASSY: a Clock Analysis System for Rapid Prototyping of Embedded Applications on MPSoCs*. Rapport de recherche RR-7918, INRIA, Mars 2012. `http://hal.inria.fr/hal-00683822`. (Cited on pages 4, 45 and 67.)

[An *et al.* 2013a] Xin An, Eric Rutten, Jean-Philippe Diguet, Nicolas le Griguer and Abdoulaye Gamatié. *Autonomic Management of Dynamically Partially Recongurable FPGA Architectures Using Discrete Control*. In In Proc. of the 10th International Conference on Autonomic Computing (ICAC'13), june 2013. (Cited on pages 4 and 70.)

[An *et al.* 2013b] Xin An, Eric Rutten, Jean-Philippe Diguet, Nicolas Le Griguer and Abdoulaye Gamatié. *Autonomic Management of Reconfigurable Embedded Systems using Discrete Control: Application to FPGA*. Rapport technique RR-8308, INRIA, Mai 2013. (Cited on pages 4 and 70.)

[An *et al.* 2013c] Xin An, Eric Rutten, Jean-Philippe Diguet, Nicolas le Griguer and Abdoulaye Gamatié. *Discrete Control for Reconfigurable FPGA-based Embedded Systems*. In In Proc. of the 4th IFAC Workshop on Dependable Control of Discrete Systems (DCDS'13), 2013. (Cited on pages 4 and 70.)

[André & Mallet 2008] Charles André and Frédéric Mallet. *Clock Constraints in UML/MARTE CCSL*. Research Report RR-6540, INRIA, 2008. http://hal.inria.fr/inria-00280941. (Cited on page 29.)

[André 1996] Charles André. *Representation and Analysis of Reactive Behaviors: A Synchronous Approach*. In Computational Engineering in Systems Applications, CESA, pages 19–29, 1996. (Cited on page 31.)

[Artieri *et al.* 2003] A. Artieri, V. D'Alto, R. Chesson, M. Hopkins and M. C. Rossi. *Nomadik-Open Multimedia Platform for Next Generation Mobile Devices*. Rapport technique TA305, STMicroelectronics, 2003. (Cited on pages 1 and 11.)

[Auer *et al.* 2009] A. Auer, J. Dingel and K. Rudie. *Concurrency control generation for dynamic threads using Discrete-Event Systems*. In Communication, Control, and Computing, 2009. Allerton 2009. 47th Annual Allerton Conference on, pages 927–934, 2009. (Cited on page 38.)

[Bailey & Martin 2010] Brian Bailey and Grant Martin. Esl models and their application: Electronic system level design and verification in practice. Springer Publishing Company, Incorporated, 2010. (Cited on page 18.)

[Balarin *et al.* 2003] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone and A. Sangiovanni-Vincentelli. *Metropolis: an integrated electronic system design environment*. Computer, vol. 36, no. 4, pages 45–52, 2003. (Cited on pages 18, 111 and 116.)

[Benveniste *et al.* 2003] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic and R. de Simone. *The synchronous languages 12 years later*. Proceedings of the IEEE, vol. 91, no. 1, pages 64 – 83, jan 2003. (Cited on pages 19, 27, 67 and 113.)

[Bjerregaard & Mahadevan 2006] Tobias Bjerregaard and Shankar Mahadevan. *A survey of research and practices of Network-on-chip*. ACM Comput. Surv., vol. 38, no. 1, Juin 2006. (Cited on page 12.)

[Blakowski & Steinmetz 1996] Gerold Blakowski and Ralf Steinmetz. *A media synchronization survey: reference model, specification and case studies*. IEEE journal on selected area in communications, 1996. (Cited on page 97.)

[Borkar *et al.* 2005] S. Y. Borkar, P. Dubey, K. C. Kahn, D. J. Kuck, H. Mulder, S. S. Pawlowski and J. R. Rattner. *Platform 2015: Intel processor and platform evolution for the next decade*. Rapport technique, Intel, 2005. http://epic.hpi.uni-potsdam.de/pub/Home/TrendsAndConceptsII2010/HW_Trends_borkar_2015.pdf. (Cited on page 1.)

[Bouhadiba *et al.* 2011] Tayeb Bouhadiba, Quentin Sabah, Gwenaël Delaval and Eric Rutten. *Synchronous control of reconfiguration in fractal component-based systems: a case study.* In Proceedings of the ninth ACM international conference on Embedded software, EMSOFT '11, pages 309–318, 2011. (Cited on page 38.)

[Boukhechem 2008] S. Boukhechem. *Contribution à la mise en place d'une plateforme open-source MPSoC sous SystemC pour la Co-simulation d'architectures hétérogènes.* PhD thesis, Université de BOURGOGNE, 2008. (Cited on page 14.)

[Boumedien 2011] Sarra Boumedien. Comparaison d'une approche symbolique et systemc pour la simulation d'applications multimedia. Master's thesis, L'Ecole Nationale d'Ingénieurs de Sfax, sep 2011. (Cited on pages 59 and 60.)

[Brunette *et al.* 2009] Christian Brunette, Jean-Pierre Talpin, Abdoulaye Gamatié and Thierry Gautier. *A metamodel for the design of polychronous systems.* J. Log. Algebr. Program., vol. 78, no. 4, pages 233–259, 2009. (Cited on page 32.)

[Burd & Brodersen 2002] Thomas D. Burd and Robert W. Brodersen. Energy efficient microprocessor design. Springer, 2002. (Cited on page 11.)

[Byna & Sun 2011] Surendra Byna and Xian-He Sun. *Special issue on Data-Intensive Computing.* Journal of Parallel and Distributed Computing, vol. 71, no. 2, pages 143 – 144, 2011. (Cited on page 1.)

[Cai & Gajski 2003] Lukai Cai and Daniel Gajski. *Transaction level modeling: an overview.* In Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES+ISSS '03, pages 19–24. ACM, 2003. (Cited on page 2.)

[Cannella *et al.* 2011] E. Cannella, L. Di Gregorio, L. Fiorin, M. Lindwer, P. Meloni, O. Neugebauer and A. Pimentel. *Towards an ESL design framework for adaptive and fault-tolerant MPSoCs: MADNESS or not?* In Proceedings of the 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia), pages 120–129, 2011. (Cited on page 13.)

[Chen & Kuo 2007] Jian-Jia Chen and Chin-Fu Kuo. *Energy-Efficient Scheduling for Real-Time Systems on Dynamic Voltage Scaling (DVS) Platforms.* In Embedded and Real-Time Computing Systems and Applications (RTCSA) 2007. 13th IEEE International Conference on, pages 28–38, aug. 2007. (Cited on page 48.)

[Chen *et al.* 2004] Rong Chen, Marco Sgroi, Luciano Lavagno, Grant Martin, Alberto Sangiovanni-Vincentelli and Jan Rabaey. Uml and platform-based design. 2004. (Cited on page 18.)

[Chiu 2000] Ge-Ming Chiu. *The odd-even turn model for adaptive routing.* Parallel and Distributed Systems, IEEE Transactions on, vol. 11, no. 7, pages 729–738, 2000. (Cited on page 12.)

[Colaço *et al.* 2005] Jean-Louis Colaço, Bruno Pagano and Marc Pouzet. *A conservative extension of synchronous data-flow with state machines.* In Proceedings of the 5th ACM international conference on Embedded software, EMSOFT '05, pages 173–182, 2005. (Cited on page 32.)

[Culler *et al.* 1999] D.E. Culler, J.P. Singh and A. Gupta. Parallel computer architecture: A hardware/software approach. Morgan Kaufmann Publishers, 1999. (Cited on page 10.)

[Dahmoune & Johnston 2010] O. Dahmoune and R. de B. Johnston. *Applying Model-Checking to Post-Silicon-Verification: Bridging the Specification-Realisation Gap.* In Proc. of the Conference on Reconfigurable Computing and FPGAs (RECONFIG'10), pages 73–78, 2010. (Cited on page 22.)

[Deb *et al.* 2000] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan. *A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II.* IEEE Transactions on Evolutionary Computation, vol. 6, pages 182–197, 2000. (Cited on page 56.)

[Delaval *et al.* 2010] G. Delaval, H. Marchand and E. Rutten. *Contracts for modular discrete controller synthesis.* In Conf. on Languages, Compilers, and Tools for Embedded Systems, pages 57–66, 2010. (Cited on pages 37, 38, 94 and 117.)

[Densmore & Passerone 2006] D. Densmore and R. Passerone. *A Platform-Based Taxonomy for ESL Design.* Design Test of Computers, IEEE, vol. 23, no. 5, pages 359–374, 2006. (Cited on page 17.)

[Drusinsky & Harel 1989] D. Drusinsky and D. Harel. *Using statecharts for hardware description and synthesis.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 8, no. 7, pages 798–807, 1989. (Cited on page 31.)

[Dumitrescu *et al.* 2010] E. Dumitrescu, A. Girault, H. Marchand and E. Rutten. *Multicriteria optimal discrete controller synthesis for fault-tolerant tasks.* In Workshop on Discrete Event Systems, Sept. 2010. (Cited on pages 33, 37, 38 and 81.)

[Durillo & Nebro 2011] Juan J. Durillo and Antonio J. Nebro. *jMetal: A Java framework for multi-objective optimization.* Advances in Engineering Software, vol. 42, pages 760–771, 2011. (Cited on page 56.)

[Edwards *et al.* 1997] S. Edwards, L. Lavagno, E.A. Lee and A. Sangiovanni-Vincentelli. *Design of embedded systems: formal models, validation, and synthesis.* Proceedings of the IEEE, vol. 85, no. 3, pages 366–390, 1997. (Cited on page 2.)

[Erbas *et al.* 2007] Cagkan Erbas, Andy D. Pimentel, Mark Thompson and Simon Polstra. *A framework for system-level modeling and simulation of embedded systems architectures.* EURASIP Journal on Embedded Systems, vol. 2007, pages 2–2, January 2007. (Cited on pages 67, 111, 114 and 116.)

[Eustache & Diguet 2008] Y. Eustache and J.-P. Diguet. *Specification and OS-based implementation of self-adaptive, hardware/software embedded systems.* In Proceedings of the 6th International conference on Hardware/Software codesign and system synthesis (CODES/ISSS'08), 2008. (Cited on pages 71, 94 and 115.)

[Gajski & Kuhn 1983] D.D. Gajski and R.H. Kuhn. *Guest Editors' Introduction: New VLSI Tools.* Computer, vol. 16, no. 12, pages 11–14, 1983. (Cited on page 13.)

[Gamatié *et al.* 2009] Abdoulaye Gamatié, Huafeng YU, Gwenaël Delaval and Éric Rutten. *A case study on controller synthesis for data-intensive embedded systems.* In Proc. 6th IEEE Int. Conf. on Embedded Software and Systems, ICESS'09, pages 75–82, 2009. (Cited on page 38.)

[Gamatié *et al.* 2011] Abdoulaye Gamatié, Sebastien Le Beux, Eric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet and Jean-Luc Dekeyser. *A Model-Driven Design Framework for Massively Parallel Embedded Systems.* ACM Trans. Embedded Comput. Syst., vol. 10, no. 4, page 39, 2011. (Cited on pages 4, 30, 46, 105 and 115.)

[Gamatié 2010] Abdoulaye Gamatié. Designing embedded systems with the SIGNAL programming language: synchronous, reactive specification. Springer New York, 2010. (Cited on page 29.)

[Gamatié 2012] Abdoulaye Gamatié. *Design of Streaming Applications on MPSoCs using Abstract Clocks.* In Design, Automation and Test in Europe Conference (DATE'2012), Dresden, Germany, 2012. (Cited on page 67.)

[Gaudin & Nixon 2012] Benoit Gaudin and Paddy Nixon. *Supervisory control for software runtime exception avoidance.* In Proceedings of the Fifth International Conference on Computer Science and Software Engineering, pages 109–112, 2012. (Cited on page 38.)

[Geilen & Basten 2010] M.C.W. Geilen and A.A. Basten. Handbook of signal processing systems, chapitre Kahn Process Networks and a Reactive Extension, pages 967–1006. Springer, Berlin, 2010. (Cited on page 26.)

[Gerstlauer *et al.* 2009] A. Gerstlauer, C. Haubelt, A.D. Pimentel, T.P. Stefanov, D.D. Gajski and J. Teich. *Electronic System-Level Synthesis Methodologies.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 28, no. 10, pages 1517–1530, 2009. (Cited on pages 13, 14 and 17.)

[Ghaffari *et al.* 2007] F. Ghaffari, M. Auguin, M. Abid and M.B. Ben Jemaa. *Dynamic and On-Line Design Space Exploration for Reconfigurable Architectures.* In Transactions on High-Performance Embedded Architectures and Compilers I, volume 4050, pages 179–193. Springer Berlin / Heidelberg, 2007. (Cited on pages 21, 111 and 115.)

[Ghamarian *et al.* 2006] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen and M. R. Mousavi. *Throughput analysis of synchronous data flow graphs.* In In Proc. of ACSD'06, IEEE, pages 25–34, 2006. (Cited on pages 27, 67 and 114.)

[Girault & Rutten 2009] Alain Girault and Éric Rutten. *Automating the addition of fault tolerance with discrete controller synthesis.* Form. Methods Syst. Des., vol. 35, no. 2, pages 190–225, Octobre 2009. (Cited on pages 36 and 38.)

[Gohringer *et al.* 2008] D. Gohringer, M.Hubner, V.Schatz and J.Becker. *Runtime adaptive multi-processor system-on-chip: RAMPSoC.* In Symp. on Parallel & Distributed Processing, pages 1–7, April 2008. (Cited on pages 21, 22 and 70.)

[Gries 2003] Matthias Gries. *Methods for Evaluating and Covering the Design Space during Early Design Development.* Integration, the VLSI Journal, vol. 38, pages 131–183, 2003. (Cited on page 15.)

[Guernic *et al.* 2002] Paul Le Guernic, Jean pierre Talpin and Jean-Christophe Le Lann. *Polychrony for System Design.* Journal for Circuits, Systems and Computers, vol. 12, pages 261–304, 2002. (Cited on page 29.)

[Gueye *et al.* 2012] Soguy Mak-Karé Gueye, Noël de Palma and Eric Rutten. *Coordinating Energy-aware Administration Loops using Discrete Control*. In Proc. of the Eighth International Conference on Autonomic and Autonomous Systems, ICAS 2012, St. Maarten, Netherlands Antilles, Mars 2012. (Cited on page 38.)

[Guillet *et al.* 2012] Sébastien Guillet, Florent de Lamotte, Nicolas Le Griguer, Eric Rutten, Guy Gogniat and Jean-Philippe Diguet. *Designing formal reconfiguration control using UML/MARTE*. In Reconfigurable and Communication Centric Systems-on-Chip, ReCoSoC, 2012. (Cited on pages 38, 94 and 115.)

[Guillet 2012] Sébastien Guillet. *Modélisation et contrôle de la reconfiguration, application aux architectures dynamiquement reconfigurables*. PhD thesis, Université de Bretagne Sud, 2012. (Cited on pages 94 and 117.)

[Ha *et al.* 2008] Soonhoi Ha, Sungchan Kim, Choonseung Lee, Youngmin Yi, Seongnam Kwon and Young-Pyo Joo. *PeaCE: A hardware-software codesign environment for multimedia embedded systems*. ACM Trans. Des. Autom. Electron. Syst., vol. 12, no. 3, pages 24:1–24:25, Mai 2008. (Cited on page 18.)

[Haid *et al.* 2009] Wolfgang Haid, Kai Huang, Iuliana Bacivarov and Lothar Thiele. *Multiprocessor SoC Software Design Flows*. IEEE Signal Processing Magazine, vol. 26, no. 6, pages 64–71, Novembre 2009. (Cited on page 19.)

[Halbwachs *et al.* 1991] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud. *The synchronous dataflow programming language LUSTRE*. In Proceedings of the IEEE, pages 1305–1320, 1991. (Cited on page 27.)

[Halbwachs 1993] Nicolas Halbwachs. Synchronous programming of reactive systems. Kluwer Academic Pub., 1993. (Cited on pages 24, 25, 26 and 31.)

[Harel 1987] David Harel. *Statecharts: A visual formalism for complex systems*. Sci. Comput. Program., vol. 8, no. 3, pages 231–274, Juin 1987. (Cited on page 31.)

[Hedde *et al.* 2009] D. Hedde, P.-H. Horrein, F. Petrot, R. Rolland and F. Rousseau. *A MPSoC Prototyping Platform for Flexible Radio Applications*. In Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on, pages 559–566, 2009. (Cited on page 18.)

[Hellerstein *et al.* 2004] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh and Dawn M. Tilbury. Feedback control of computing systems. John Wiley & Sons, 2004. (Cited on page 38.)

[Huebscher & McCann 2008] Markus C. Huebscher and Julie A. McCann. *A survey of autonomic computing-degrees, models, and applications*. ACM Comput. Surv., vol. 40, no. 3, pages 7:1–7:28, Août 2008. (Cited on pages 23 and 118.)

[IBM 2006] IBM. *An Architectural Blueprint for Autonomic Computing*. Rapport technique, 2006. `http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf`. (Cited on pages 22 and 23.)

[Iordache & Antsaklis 2009] Marian V. Iordache and Panos J. Antsaklis. *Synthesis of Concurrent Programs Based on Supervisory Control*, 2009. (Cited on page 38.)

[Jovanović *et al.* 2008] S. Jovanović, C. Tanougast and S. Weber. *A New Self-managing Hardware Design Approach for FPGA-Based Reconfigurable Systems.* In Reconfigurable Computing: Architectures, Tools and Applications, volume 4943 of *LNCS*. 2008. (Cited on pages 94 and 115.)

[Kahn 1974] Gilles Kahn. *The Semantics of Simple Language for Parallel Programming.* In IFIP Congress, pages 471–475, 1974. (Cited on pages 19 and 26.)

[Kangas *et al.* 2006] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen, Jouni Riihimäki and Kimmo Kuusilinna. *UML-based multiprocessor SoC design framework.* ACM Trans. Embed. Comput. Syst., vol. 5, no. 2, pages 281–320, Mai 2006. (Cited on pages 18 and 26.)

[Kathuria *et al.* 2011] Jagrit Kathuria, M. Ayoubkhan and Arti Noor. *A Review of Clock Gating Techniques.* MIT International Journal of Electronics and Communication Engineering, 2011. (Cited on page 11.)

[Keinert *et al.* 2009] Joachim Keinert, Martin Streubuhr, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jurgen Teich and Michael Meredith. *SystemCoDesigner-an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications.* ACM Trans. Des. Autom. Electron. Syst., vol. 14, no. 1, pages 1:1–1:23, Janvier 2009. (Cited on page 18.)

[Kephart & Chess 2003] J.O. Kephart and D.M. Chess. *The vision of autonomic computing.* Computer, vol. 36, no. 1, pages 41–50, 2003. (Cited on page 22.)

[Kienhuis *et al.* 2002] Bart Kienhuis, Ed F. Deprettere, Pieter van der Wolf and Kees A. Vissers. *A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach.* In Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS, pages 18–37, 2002. (Cited on page 16.)

[Klaiber 2000] A. Klaiber. *The technology behind Crusoe processors.* Rapport technique, Transmeta, 2000. `http://www.cs.ucf.edu/~lboloni/Teaching/EEL5708_2004/slides/paper_aklaiber_19jan00.pdf`. (Cited on page 11.)

[Le Sueur & Heiser 2010] Etienne Le Sueur and Gernot Heiser. *Dynamic voltage and frequency scaling: the laws of diminishing returns.* In Proceedings of the 2010 international conference on Power aware computing and systems, HotPower'10, pages 1–8, 2010. (Cited on page 11.)

[Lee & Messerschmitt 1987] Edward Ashford Lee and David G. Messerschmitt. *Static scheduling of synchronous data flow programs for digital signal processing.* IEEE Trans. Comput., vol. 36, pages 24–35, January 1987. (Cited on pages 19, 26 and 51.)

[Lee & Sangiovanni-vincentelli 1998] Edward A. Lee and Alberto Sangiovanni-vincentelli. *A Framework for Comparing Models of Computation.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, pages 1217–1229, 1998. (Cited on pages 17 and 47.)

[Lee *et al.* 2006] Hyung Gyu Lee, U.Y. Ogras, R. Marculescu and N. Chang. *Design space exploration and prototyping for on-chip multimedia applications.* In Design Automation Conference, 2006 43rd ACM/IEEE, pages 137 –142, 0-0 2006. (Cited on page 2.)

[Lee *et al.* 2008] Hyung Gyu Lee, Naehyuck Chang, Umit Y. Ogras and Radu Marculescu. *On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches*. ACM Trans. Des. Autom. Electron. Syst., vol. 12, no. 3, pages 23:1–23:20, Mai 2008. (Cited on page 12.)

[Lee 2003] Edward A. Lee. *Overview of the Ptolemy Project*. Rapport technique, University of California, Berkeley, 2003. http://www.ptolemy.eecs.berkeley.edu/publications/papers/03/overview/overview03.pdf. (Cited on page 18.)

[Liu *et al.* 2008] Cong Liu, Alex Kondratyev, Yosinori Watanabe, Jörg Desel and Alberto Sangiovanni-Vincentelli. *Schedulability Analysis of Petri Nets Based on Structural Properties*. Fundam. Inf., vol. 86, no. 3, pages 325–341, Août 2008. (Cited on page 38.)

[Maggio *et al.* 2012] Martina Maggio, Henry Hoffmann, Alessandro V. Papadopoulos, Jacopo Panerati, Marco D. Santambrogio, Anant Agarwal and Alberto Leva. *Comparison of Decision-Making Strategies for Self-Optimization in Autonomic Computing Systems*. ACM Trans. Auton. Adapt. Syst., vol. 7, no. 4, pages 36:1–36:32, 2012. (Cited on pages 23, 94 and 115.)

[Majer *et al.* 2007] Mateusz Majer, Jürgen Teich, Ali Ahmadinia and Christophe Bobda. *The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer*. The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, vol. 47, pages 15–31, 2007. (Cited on pages 94 and 115.)

[Mallet 2011] Frédéric Mallet. Logical Time @ Work for the Modeling and Analysis of Embedded Systems. LAP LAMBERT Academic Publishing, Janvier 2011. (Cited on page 29.)

[Maraninchi & Rémond 2001] F. Maraninchi and Y. Rémond. *Argos: an Automaton-Based Synchronous Language*. Computer Languages, no. 27, pages 61–92, 2001. (Cited on page 31.)

[Maraninchi & Rémond 2003] Florence Maraninchi and Yann Rémond. *Mode-automata: a new domain-specific construct for the development of safe critical systems*. Sci. Comput. Program., vol. 46, no. 3, pages 219–254, Mars 2003. (Cited on pages 31 and 32.)

[Marchand & Samaan 2000] H. Marchand and M. Samaan. *Incremental design of a power transformer station controller using a controller synthesis methodology*. IEEE Trans. on Soft. Eng., vol. 26, no. 8, pages 729 –741, 2000. (Cited on pages 4, 36, 37, 81, 93 and 114.)

[Marchand *et al.* 2000] Hervé Marchand, Patricia Bournai, Michel Le Borgne and Paul Le Guernic. *Synthesis of Discrete-Event Controllers based on the Signal Environment*. Discrete Event Dynamic System: Theory and Applications, vol. 10, no. 4, Octobre 2000. (Cited on page 38.)

[Marwedel *et al.* 2011] P. Marwedel, J. Teich, G. Kouveli, I. Bacivarov, L. Thiele, S. Ha, C. Lee, Q. Xu and L. Huang. *Mapping of applications to MPSoCs*. In Proc. of the 7th international conference on Hardware/software codesign and system synthesis, CODES+ISSS'11, pages 109–118, 2011. (Cited on page 17.)

[Marwedel 2011] Peter Marwedel. Embedded system design. Springer London, Limited, 2011. (Cited on pages 10, 11, 15 and 26.)

[May *et al.* 2010] Matthias May, Norbert Wehn, Abdelmajid Bouajila, Johannes Zeppenfeld, Walter Stechele, Andreas Herkersdorf, Daniel Ziener and Jürgen Teich. *A rapid prototyping system for error-resilient multi-processor systems-on-chip.* In Proceedings of the Conference on Design, Automation and Test in Europe, DATE '10, pages 375–380. European Design and Automation Association, 2010. (Cited on page 2.)

[Milutinovic *et al.* 2009] Aleksandar Milutinovic, Anca Molnos, Kees Goossens and Gerard J.M. Smit. *Dynamic Voltage and Frequency Scaling and Adaptive Body Biasing for Active and Leakage Power Reduction in MPSoC: a Literature Overview.* In Proceedings of the Program for Research on Ingrated Systems and Circuits, ProRISC 2009, pages 488–495, November 2009. (Cited on page 11.)

[Mirza-Aghatabar *et al.* 2007] M. Mirza-Aghatabar, S. Koohi, S. Hessabi and M. Pedram. *An Empirical Investigation of Mesh and Torus NoC Topologies Under Different Routing Algorithms and Traffic Models.* In Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on, pages 19–26, 2007. (Cited on pages 12 and 13.)

[Mitra *et al.* 2010] S. Mitra, S. A. Seshia and N. Nicolici. *Post-silicon validation opportunities, challenges and recent advances.* In Proc. of the 47th Design Automation Conference, pages 12–17, 2010. (Cited on page 21.)

[Moadeli *et al.* 2007] Mahmoud Moadeli, Ali Shahrabi, Wim Vanderbauwhede and Mohamed Ould-Khaoua. *An Analytical Performance Model for the Spidergon NoC.* In Proceedings of the 21st International Conference on Advanced Networking and Applications, AINA '07, pages 1014–1021, 2007. (Cited on page 12.)

[Nollet *et al.* 2008] V. Nollet, P. Avasare, H. Eeckhaut, D. Verkest and H. Corporaal. *Run-Time Management of a MPSoC Containing FPGA Fabric Tiles.* Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 16, no. 1, pages 24 –33, jan. 2008. (Cited on page 21.)

[Nollet 2008] V. Nollet. *Run-time management for future MPSoC platforms.* PhD thesis, Technische Universiteit Eindhoven, 2008. (Cited on page 10.)

[Object Management Group 2013a] Object Management Group. *A UML Profile for MARTE*, 2013. `http://www.omgmarte.org`. (Cited on pages 29, 45, 46, 73 and 96.)

[Object Management Group 2013b] Object Management Group. *Unified Modeling Language*, 2013. `http://www.uml.org/`. (Cited on page 29.)

[Parashar & Hariri 2005] Manish Parashar and Salim Hariri. *Autonomic computing: An overview.* In Unconventional Programming Paradigms, pages 247–259. Springer Verlag, 2005. (Cited on page 22.)

[Paulin *et al.* 2002] P.G. Paulin, C. Pilkington and E. Bensoudane. *StepNP: a system-level exploration platform for network processors.* Design Test of Computers, IEEE, vol. 19, no. 6, pages 17–26, 2002. (Cited on page 18.)

[Petrot *et al.* 2011] F. Petrot, M. Gligor, M.-M. Hamayun, Hao Shen, N. Fournel and P. Gerin. *On MPSoC Software Execution at the Transaction Level.* Design Test of Computers, IEEE, vol. 28, no. 3, pages 32–43, 2011. (Cited on page 18.)

[Potop-Butucaru *et al.* 2005] Dumitru Potop-Butucaru, Robert de Simone and Jean-Pierre Talpin. *The Synchronous Hypothesis and Synchronous Languages*. 2005. (Cited on page 27.)

[Quadri *et al.* 2010a] Imran Rafiq Quadri, Abdoulaye Gamatié, Pierre Boulet and Jean-Luc Dekeyser. *Modeling of configurations for embedded system implementations in* MARTE. In 1st workshop on Model Based Engineering for Embedded Systems Design - Design, Automation and Test in Europe (DATE 2010), Dresden Germany, 2010. (Cited on page 98.)

[Quadri *et al.* 2010b] Imran Rafiq Quadri, Huafeng Yu, Abdoulaye Gamatié, Eric Rutten, Samy Meftali and Jean-Luc Dekeyser. *Targeting reconfigurable FPGA based SoCs using the UML MARTE profile: from high abstraction levels to code generation*. IJES, vol. 4, no. 3/4, pages 204–224, 2010. (Cited on pages 22, 94 and 115.)

[Ramadge & Wonham 1989] P.J. Ramadge and W.M. Wonham. *On the Supervisory Control of Discrete Event Systems*. Proc. IEEE, vol. 77, no. 1, Janvier 1989. (Cited on pages 4, 35, 93 and 114.)

[Robert & Perrier 2010] Thomas Robert and Vincent Perrier. *A UML design flow aimed at embedded systems*, 2010. http://www.techdesignforums.com/practice/technique/a-uml-design-flow-aimed-at-embedded-systems. (Cited on page 18.)

[Rowe & Smith 1993] Lawrence A. Rowe and Brian C. Smith. *A continuous media player*. In Network and Operating System Support for Digital Audio and Video, pages 376–386. Springer Berlin / Heidelberg, 1993. (Cited on page 97.)

[Santambrogio 2010] Marco D. Santambrogio. *From reconfigurable architectures to self-adaptive autonomic systems*. IJES, vol. 4, no. 3/4, pages 172–181, 2010. (Cited on pages 22 and 70.)

[Schaefer & Poetzsch-Heffter 2009] Ina Schaefer and Arnd Poetzsch-Heffter. *Model-based verification of adaptive embedded systems under environment constraints*. SIGBED Rev., vol. 6, no. 3, pages 9:1–9:4, Octobre 2009. (Cited on page 21.)

[Schor *et al.* 2012] Lars Schor, Iuliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin haeng Kang and Lothar Thiele. *Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems*. In Proc. International Conference on Compilers Architecture and Synthesis for Embedded Systems (CASES), pages 71–80, Tampere, Finland, 2012. ACM. (Cited on pages 1, 19, 21, 67, 111, 114 and 115.)

[Schranzhofer *et al.* 2010] A. Schranzhofer, Jian-Jian Chen and L. Thiele. *Dynamic Power-Aware Mapping of Applications onto Heterogeneous MPSoC Platforms*. Industrial Informatics, IEEE Transactions on, vol. 6, no. 4, pages 692 –707, nov. 2010. (Cited on pages 21, 111 and 115.)

[Siala & Saoud 2011] Med Aymen Siala and Slim Ben Saoud. *A Survey on Existing MPSOCs Architectures*. International Journal of Computer Applications, vol. 19, no. 3, pages 28–41, April 2011. Published by Foundation of Computer Science. (Cited on pages 9 and 12.)

[Sigdel *et al.* 2012] K. Sigdel, C. Galuzzi, K. Bertels, M. Thompson and A.D. Pimentel. *Evaluation of runtime task mapping using the rSesame framework*. Int. J. Reconfig. Comput., vol. 2012, Janvier 2012. (Cited on pages 111 and 116.)

[Sigdel 2011] Kamana Sigdel. *System-level design space exploration of reconfigurable architectures*. PhD thesis, Delft University of Technology, 2011. (Cited on pages 67 and 114.)

[sim 2013] *The SimpleScalar-Arm Power Modeling Project*, 2013. http://web.eecs.umich.edu/~panalyzer. (Cited on page 66.)

[Singh *et al.* 2011] Amit Kumar Singh, Akash Kumar and Thambipillai Srikanthan. *A hybrid strategy for mapping multiple throughput-constrained applications on MPSoCs*. In Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems, CASES '11, pages 175–184, 2011. (Cited on pages 21, 111 and 115.)

[Sironi *et al.* 2010] F. Sironi, M. Triverio, H. Hoffmann, M. Maggio and M.D. Santambrogio. *Self-Aware Adaptation in FPGA-based Systems*. In Field Programmable Logic and Applications, 2010. (Cited on pages 94 and 115.)

[Smit *et al.* 2005] L.T. Smit, J.L. Hurink and G.J.M. Smit. *Run-time Mapping of Applications to a Heterogeneous SoC*. In Proc. of the International Symposium on System-on-Chip, pages 78–81, nov. 2005. (Cited on pages 21, 111 and 115.)

[SoClib 2012] SoClib. *The SoClib Project*, 2012. http://www.soclib.fr. (Cited on pages 18, 58, 59 and 66.)

[Stephen A. Edwards 2001] Stephen A. Edwards. *Dataflow languages*, 2001. http://www.google.fr/url?sa=t&rct=j&q=dataflowurce=web&cd=1&ved=0CDUQFjAA&url=httpFclassesoDYDg&usg=AFQjCNEL9VSbgEYIbMlVY826Hz7E5yPozw&sig2=z6k1k29NROjRpzslKj6zvQ&bvm=bv.47883778,d.bGE. (Cited on page 26.)

[Stuijk *et al.* 2006a] S. Stuijk, M. Geilen and T. Basten. *SDF^3: SDF For Free*. In Sixth International Conference on Application of Concurrency to System Design (ACSD), pages 276–278, 2006. (Cited on pages 27, 64 and 116.)

[Stuijk *et al.* 2006b] Sander Stuijk, Marc Geilen and Twan Basten. *Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs*. In DESIGN AUTOMATION CONFERRENCE, PROC. ACM, pages 899–904, 2006. (Cited on page 2.)

[Stuijk *et al.* 2008] Sander Stuijk, Marc Geilen and Twan Basten. *Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs*. IEEE Trans. Comput., vol. 57, no. 10, pages 1331–1345, oct 2008. (Cited on pages 27, 58, 61, 62, 63, 64, 67, 114 and 120.)

[Stuijk *et al.* 2010] S. Stuijk, M. Geilen and T. Basten. *A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour*. In 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD), 2010, pages 548–555, sept. 2010. (Cited on pages 47, 65, 67, 111, 114 and 115.)

[Stuijk *et al.* 2011] Sander Stuijk, Marc Geilen, Bart D. Theelen and Twan Basten. *Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications*. In Luigi Carro and Andy D. Pimentel, editeurs, ICSAMOS, pages 404–411, 2011. (Cited on pages 19, 67, 111, 114 and 116.)

[Stuijk 2007] Sander Stuijk. *Predictable mapping of streaming applications on multiprocessors*. PhD thesis, TU Eindhoven, 2007. (Cited on pages 19, 58, 62, 64, 65, 67 and 114.)

[Tammemäe & Ellervee] Kalle Tammemäe and Peeter Ellervee. *Abstraction levels: Y-chart*. http://mini.li.ttu.ee/~lrv/IAY3714/synt-meth.pdf. (Cited on pages 14 and 118.)

[Thiele *et al.* 2007] L. Thiele, I. Bacivarov, W. Haid and Kai Huang. *Mapping Applications to Tiled Multiprocessor Embedded Systems*. In Application of Concurrency to System Design, 2007. ACSD 2007. Seventh International Conference on, pages 29–40, july 2007. (Cited on page 26.)

[Thompson *et al.* 2007] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra and Ed F. Deprettere. *A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs*. In CODES+ISSS'07, pages 9–14, New York, NY, USA, 2007. ACM. (Cited on pages 2, 17 and 26.)

[Wang *et al.*] Yin Wang, Hyoun Kyu Cho, Hongwei Liao, Ahmed Nazeem, Terence P. Kelly, Stéphane Lafortune, Scott Mahlke and Spyros A. Reveliotis. *Supervisory control of software execution for failure avoidance: Experience from the Gadara project*. In Proceedings of the 10th International Workshop on Discrete Event Systems (WODES'10). (Cited on page 38.)

[Wang *et al.* 2009] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur and Scott Mahlke. *The theory of deadlock avoidance via discrete control*. SIGPLAN Not., vol. 44, no. 1, pages 252–263, Janvier 2009. (Cited on page 38.)

[Wildermann *et al.* 2010] Stefan Wildermann, Andreas Oetken, Jürgen Teich and Zoran Salcic. *Self-organizing computer vision for robust object tracking in smart cameras*. In Proceedings of the 7th international conference on Autonomic and trusted computing, ATC'10, pages 1–16, 2010. (Cited on page 47.)

[Wolf *et al.* 2008] Wayne Wolf, Ahmed Amine Jerraya and Grant Martin. *Multiprocessor System-on-Chip (MPSoC) Technology*. IEEE Trans. on CAD of Integrated Circuits and Systems, vol. 27, no. 10, pages 1701–1713, 2008. (Cited on pages 1, 11, 12, 44 and 118.)

[Xilinx 2013] Xilinx. *Platform Studio and the Embedded Development Kit (EDK)*, 2013. http://www.xilinx.com/tools/platform.htm. (Cited on page 20.)

[Yang *et al.* 2009] Yang Yang, M. Geilen, T. Basten, S. Stuijk and H. Corporaal. *Exploring trade-offs between performance and resource requirements for synchronous dataflow graphs*. In Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on, pages 96–105, 2009. (Cited on page 19.)

[Yang *et al.* 2012] Y. Yang, M. Geilen, T. Basten, S. Stuijk and H. Corporaal. *Playing games with scenario- and resource-aware SDF graphs through policy iteration*. In Design, Automation Test in Europe Conference Exhibition (DATE), pages 194 – 199, march 2012. (Cited on page 21.)

[Zhu *et al.* 2010] Jun Zhu, I. Sander and A. Jantsch. *Pareto efficient design for reconfigurable streaming applications on CPU/FPGAs*. In Design, Automation Test in

Europe Conference Exhibition (DATE), 2010, pages 1035–1040, 2010. (Cited on page 19.)

[Zitzler 1999] E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, ETH Zurich, Switzerland, 1999. (Cited on pages 15, 16, 55, 56 and 118.)